VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Řízení robotické ruky pomocí vícejádrového hybridního procesoru

Usage of Multicore Hybrid CPU for Robotic Arm Control

2017 Bc. Aleš Prchal

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Aleš Prchal**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Řízení robotické ruky pomocí vícejádrového hybridního procesoru
Usage of Muticore Hybrid CPU for Robotic Arm Control

Jazyk vypracování: čeština

Zásady pro vypracování:

Navrhněte řídící systém pro řízení robotické ruky pomocí dvoujádrového hybridního procesoru iMX 6SoloX. Hlavní jádro CPU s OS Linux bude sloužit pro zajištění uživatelského rozhraní a zpětné vazby pomocí kamery. Jádro Cortex M4 bude využívat FreeRTOS a zajistí řízení robotické ruky v reálném čase. Při realizaci je potřeba klást důraz zejména na stabilitu řešení, aby při jakékoliv nestabilitě OS Linux z důvodu přetížení, nebo při nestabilitě GUI, bylo zajištěno bezpečné řízení robotické ruky v jádře M4.

1. Seznamte se s architekturou hybridního procesoru iMX 6SoloX.
2. Seznamte se s robotickou rukou a stávající aplikací pro ovládání robotické ruky.
3. Analyzujte stávající stav aplikace a navrhněte rozdělení řízení robotické ruky na část pro FreeRTOS a pro OS Linux.
4. Naprogramujte RT část aplikace pro M4 jádro ve FreeRTOSu. Zajistěte stabilitu řízení i v případech havárie aplikace v OS Linux.
5. Otestujte navržené řešení, jeho spolehlivost a stabilitu.
6. Zvažte možnosti nahrazení řídícího modulu robotické ruky přímým řízením pomocí M4. Navržené úpravy realizujte a otestuje.

Seznam doporučené odborné literatury:

[1] RTOS: http://www.freertos.org/
[2] Linux Začínáme programovat, Richard Stones, Neil Metthew, COMPUTER PRESS, ISBN 80–7226–307–2
[3] Christopher Hallinan: Embedded Linux Primer, Second Edition. Prentice Hall 2011, ISBN 0131679848
[4] Lynxmotion Servo Controller SSC-32, http://www.lynxmotion.com/p-395-ssc-32-servo-controller.aspx
[5] Lynxmotion L6AC-KT Robotic Arm, http://www.lynxmotion.com/driver.aspx?Topic=assem01#l6

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Petr Olivka, Ph.D.**

Datum zadání: 01.09.2016

Datum odevzdání: 28.04.2017

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 28. dubna 2017 .....

Rád bych na tomto místě poděkoval Ing. Petru Olivkovi, Ph.D. za cenné rady při vedení této diplomové práce.

Abstrakt

Práce se zabývá návrhem změn řídícího systému pro robotickou ruku a jejich následnou implementací. Původní řešení využívající vývojovou desku i.MX Sabre SD je nahrazeno platformou UDOO Neo. Z hlediska softwaru dochází k rozložení původního řešení na dvě samostatné komponenty: zatímco uživatelské rozhraní pro ovládání robotické ruky běží v rámci linuxové distribuce na výpočetním jádru ARM Cortex A9, samotný kód pro ovládání serv je založen na realtime operačním systému FreeRTOS a využívá jádro ARM Cortex M4. Výměna informací mezi oběma komponentami je zajištěna na úrovni meziprocesorové komunikace s využitím protokolu RPMsg. Takto upravený systém vykazuje kromě menších fyzických rozměrů i zvýšenou odolnost vůči výpadkům grafického rozhraní.

Klíčová slova: ARM i.MX6 SoloX, FreeRTOS, UDOO Neo, robotická ruka, RPMsg

Abstract

This thesis deals with an improvement of a robotic arm control system. In order to accomplish predefined goals, several modifications have been made. The original development board has been replaced by the more compact UDOO Neo board while the software has been divided into two parts running on different cores. The graphical user interface is now served by the Linux operating system running on a general purpose ARM Cortex A9 core whilst the critical tasks are managed by the FreeRTOS on a specialized ARM Cortex M4 processing unit. The need of interprocessor communication is fulfilled by the use of RPMsg protocol. As a result, a more compact and robust robotic arm controller solution has been created.

Key Words: ARM i.MX6 SoloX, FreeRTOS, UDOO Neo, robotic arm, RPMsg

Obsah

Seznam použitých zkratek a symbolů **9**

Seznam obrázků **10**

Seznam tabulek **11**

1 Úvod **12**

2 Seznámení s výchozím stavem **14**
2.1 Stávající hardwarové řešení 14
2.2 Aplikace RoboticArm 16

3 Popis modifikovaného řešení **19**
3.1 Hardwarová platforma 20
3.2 Softwarová komponenta pro jádro Cortex A9 24
3.2.1 Linuxová distribuce 24
3.2.2 Mechanismus meziprocesorové komunikace 28
3.2.3 Komunikační framework virtio 30
3.2.4 Úpravy softwaru 34
3.3 Softwarová komponenta pro jádro Cortex M4 36
3.3.1 FreeRTOS 36
3.3.2 Programové řešení s GPIO 38
3.3.3 Řešení s využitím PWM rozhraní 41

4 Test modifikovaného řešení **44**
4.1 Meziprocesorová komunikace 44
4.2 Restart aplikace RoboticArm 45
4.3 Generování výstupních pulsů 46
4.4 Ovládání robotické ruky 48

5 Závěr **49**

Literatura **51**

Přílohy **52**

A Obsah přiloženého média **53**

B Postup při modifikaci výchozí instalace Udoobuntu **54**
B.1 Zprovoznění RPMsg 54
B.2 Kompilace a nahrání programu pro jádro Cortex M4 58
B.3 Zprovoznění GUI aplikace RoboticArm 59

C Rozšiřující deska pro připojení serv **60**

Seznam použitých zkratek a symbolů

AMP	–	Asymmetric Multiprocessing
API	–	Application Programming Interface
BSP	–	Board Support Package
CAN	–	Controller Area Network
DPS	–	Deska plošných spojů
DTB	–	Device Tree Blob
DTS	–	Device Tree Source
EPIT	–	Enhanced Periodic Interrupt Timer
GND	–	Ground
GPIO	–	General Purpose Input/Output
IPI	–	Inter-Processor Interrupt
ISR	–	Interrupt Service Routine
LVDS	–	Low-voltage differential signaling
PWM	–	Pulse Width Modulation
RPC	–	Remote Procedure Call
RDC	–	Resource Domain Controller
RPMsg	–	Remote Processor Messaging
RTOS	–	Real-time Operating System
SoC	–	System on a Chip
SPI	–	Serial Peripheral Interface
UART	–	Universal Asynchronous Receiver/Transmitter

Seznam obrázků

1 Původní systém pro demonstraci ovládání robotické ruky. 14
2 Blokové schéma původního zapojení. 15
3 Schématické znázornění robotické ruky L6AC. 16
4 Princip nastavení polohy serva pomocí šířky ovládacího pulsu. 17
5 Ukázky aplikace RoboticArm. 18
6 Rozmístění komponent na desce UDOO Neo. 19
7 Blokové schéma navrhovaného zapojení UDOO Neo. 21
8 Blokové schéma SoC i.MX6 SoloX. 22
9 Rozdělení adresního prostoru jádra Cortex A9. 23
10 Grafická rozhraní vybraných aplikací v Udoobuntu. 26
11 Schématické znázornění RPMsg komponent. 30
12 Formát RPMsg zprávy. 30
13 Znázornění datových struktur `virtqueue` a `vring`. 31
14 Princip komunikace pomocí virtio pro jednotlivé směry. 33
15 Blokové schéma programu pro jádro Cortex M4. 37
16 Schématické znázornění úlohy pro RPMsg komunikaci. 39
17 Schématické znázornění úlohy pro ovládání serv pomocí GPIO. 40
18 Schématické znázornění úlohy pro ovládání serv pomocí PWM. 42
19 Zachycený průběh řídících pulsů. 47
20 UDOO Neo s rozšiřující deskou pro připojení serv. 48
21 RPMsg sekce v prostředí nástroje menuconfig. 55
22 Schéma rozšiřující desky pro připojení serv. 60
23 DPS rozšiřující desky. 61

Seznam tabulek

1 Zapojení serv do modulu SSC-32. 15
2 Seznam modifikovaných souborů linuxového jádra. 35
3 Konfigurace GPIO pinů. 39
4 Konfigurace PWM pinů. 41

1 Úvod

Moderní generace jednočipových systémů bývají zpravidla osazeny několika výpočetními jádry. Na základě vnitřní architektury čipu pak rozlišujeme mezi *symetrickým multiprocessingem* (SMP), v rámci něhož mají jádra rovnocenný přístup do paměti, shodnou architekturu a jsou používány jedním operačním systémem, a *asymetrickým multiprocessingem* (AMP), v němž neplatí některá z uvedených podmínek. Především v oblasti vestavěných systémů nabývá AMP v poslední době značného ohlasu, neboť umožňuje sdružit k hlavnímu výpočetnímu jádru i několik specializovaných jader pro plnění specifických úkolů. To je i případ čipu i.MX6 SoloX od společnosti Freescale (nyní NXP), jenž v sobě kromě hlavního jádra Cortex A9 obsahuje i specializovanou výpočetní jednotku Cortex M4, která je obzvláště vhodná pro nasazení v realtime prostředí. V rámci diplomové práce proto dojde k využití platformy i.MX6 SoloX spolu s asymetrickým multiprocessingem za účelem zdokonalení systému pro řízení robotické ruky.

Samotná práce je rozdělena do tří tématických celků. V kapitole 2 je popsáno stávající řešení pro ovládání robotické ruky. Původní systém se z hlediska hardwaru skládá z několika vzájemně propojených modulů; jedná se především o hlavní desku s čipem i.MX6 SoloX a řídící modul pro ovládání serv. Kromě toho je nedílnou součástí celého řešení i softwarová komponenta, jež kromě vlastní linuxové distribuce zahrnuje i aplikaci s grafickým rozhraním pro zadávání uživatelských povelů.

Navazující kapitola 3 se zabývá popisem změn, jež vedou k naplnění jednotlivých bodů zadání diplomové práce. Čtenáři je proto neprve představena nová deska UDOO Neo, s jejíž pomocí lze ovládat serva bez nutnosti externího řídícího modulu. Stejně jako v případě původní desky je srdcem navrhovaného řešení čip i.MX6 SoloX. Z tohoto důvodu se text v krátkosti věnuje popisu funkčních komponent čipu, především pak těch, které nějakým způsobem souvisí se zbývajícími částmi práce. Jelikož se na zmíněném čipu nachází dvojice výpočetních jednotek, bude autorovo úsilí směřovat k jejich efektivnímu využití. V rámci jednoho z jader (konkrétně Cortex A9) poběží operační systém Linux, který bude zajišťovat interakci s uživatelem. Druhá výpočetní jednotka (Cortex M4) pak poslouží pro realtime ovládání serv robotické ruky. Mezi oběma jádry bude nutné zajistit výměnu dat, proto se část textu věnuje popisu možností meziprocesorové komunikace.

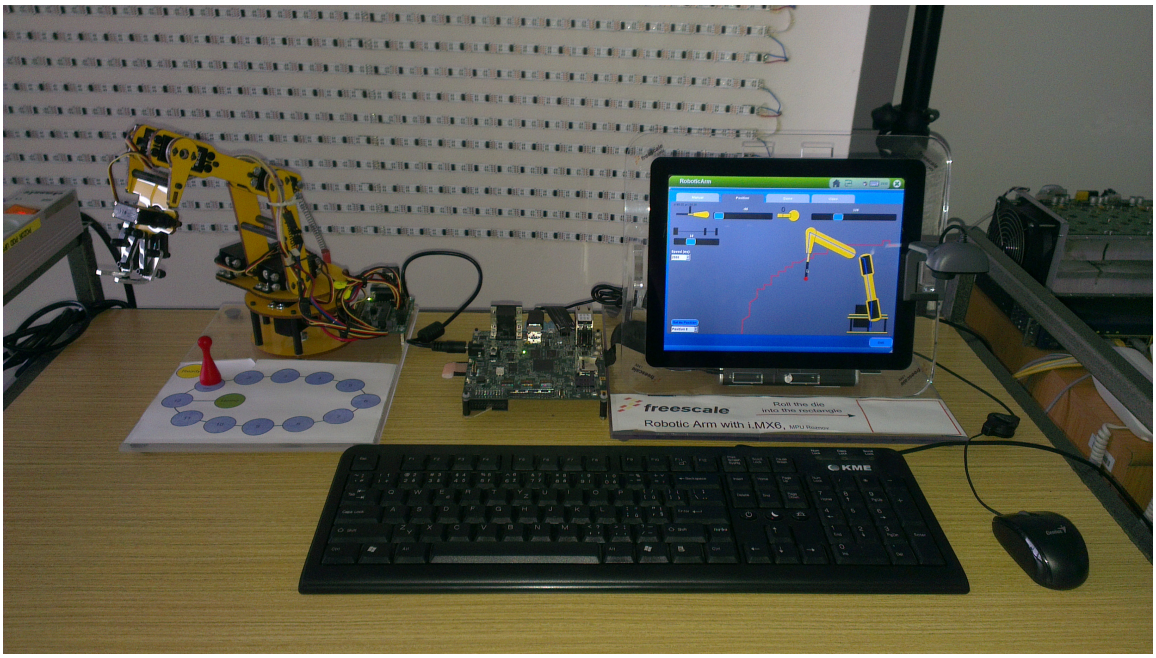
V rámci softwarové části je čtenář nejprve seznámen s linuxovou distribucí Udoobuntu, na které poběží aplikace pro uživatelské ovládání robotické ruky. Jak se vzápětí ukáže, bude kvůli zprovoznění meziprocesorové komunikace potřeba uvedenou distribuci modifikovat. Pro ovládání serv pak dojde k implementaci programu v jazyce C. Vzniklý kód bude založen na realtime operačním systému FreeRTOS a poběží na jádru Cortex M4. Úkolem programu bude přijímat požadavky na nastavení serv do nových pozic a následně je realizovat.

Ve finální části 4 je provedena série testů, v rámci nichž dochází jednak k ověření správnosti implementace, ale rovněž i stability systému při výpadku aplikace s grafickým uživatelským

rozhraním. Součástí práce je rovněž i bohatá příloha, v níž lze mj. nalézt postup, jak upravit výchozí instalaci Udoobuntu do stavu umožňujícího ovládání robotické ruky.

2 Seznámení s výchozím stavem

Hlavní cíl diplomové práce spočívá v návrhu řídícího systému pro ovládání pohybu robotické ruky. Na tomto místě je však třeba podotknout, že řešení úlohy nezačíná tzv. „na zelené louce“, nýbrž vychází z již existujícího projektu. Ten byl dodán zaměstnanci společnosti NXP ve formě ukázkové sady pro demonstraci možností výpočetní platformy i.MX6 SoloX. Uvedené řešení, jehož foto je k dispozici na obr. 1, umožňuje uživateli ovládát polohu robotické ruky pomocí dotykového displeje. Kromě toho si lze zahrát i zjednodušenou variantu hry „Člověče, nezlob se!“, v rámci níž je možné hod kostkou snímat prostřednictvím připojené webkamery.

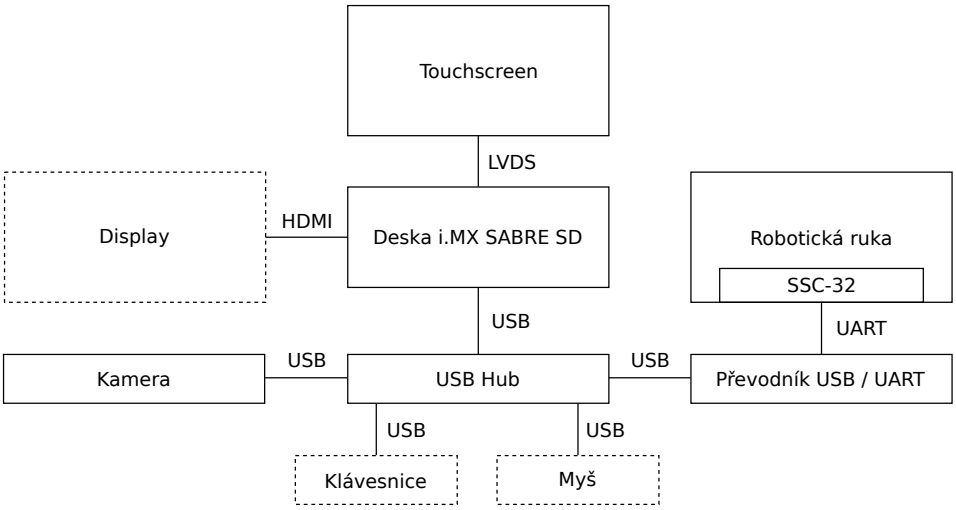


Obrázek 1: Původní systém pro demonstraci ovládání robotické ruky.

Před samotným popisem navrhovaného řešení je proto vhodné čtenáře nejprve seznámit s výchozím projektem, a to především s důrazem na mechanismus ovládání robotické ruky a na GUI aplikaci, jež má na starosti řízení celého systému.

2.1 Stávající hardwarové řešení

Původní systém se skládá z několika funkčních komponent, jejichž propojení je znázorněno na obr. 2. Centrálním prvkem je zde vývojová deska i.MX Sabre SD od firmy NXP, která je osazena čipem i.MX6 SoloX. K desce je pomocí LVDS kabelu připojen dotykový displej, jenž slouží jako primární prostředek pro zajištění interakce s uživatelem. Jelikož má centrální deska nedostatečný počet USB portů, používá se pro připojení zbývajících komponent USB hub. Do něj je připojena klávesnice s myší jakožto záložními prostředky pro ovládání, webkamera značky Genius a dále převodník mezi USB a RS232.



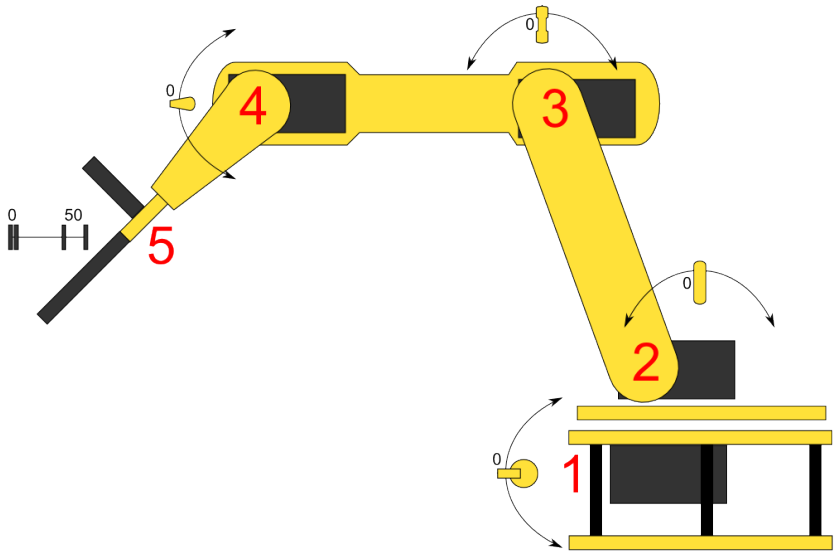
Obrázek 2: Blokové schéma původního zapojení. Čárkovaně je znázorněna záložní varianta ovládání.

Klíčovou součástí celého systému je robotická ruka Lynxmotion L6AC [1], jež je schématicky znázorněna na obr. 3. Pro změnu její polohy slouží celkem šest serv typu HS-422, která umožňují pohyb ruky v pěti stupních volnosti (vzhledem k omezenému výkonu použitého modelu serva je pro rotaci v rámci kloubu č. 2 potřeba dvou spřažených serv). Pohyb jednotlivých serv je ovládán s využitím kontroléru Lynxmotion SSC-32 [2], jemuž jsou příkazy zasílány prostřednictvím sériové linky. Z tohoto důvodu je robotická ruka jako celek propojena s vývojovou deskou pomocí již dříve zmíněného převodníku USB/RS232. Přestože použitý kontrolér umožňuje nezávisle ovládat až 32 serv, v popisovaném zapojení se využívá pouze 6 kanálů. Informace ohledně mapování jednotlivých serv na číslo kanálu nalezne čtenář v tab. 1.

Míra natočení serva je určena šířkou impulsu, který je přiveden na jeho řídící vstup. V případě výše uvedeného modelu HS-422 se používá šířka v rozmezí 500 – 2500 µs, viz obr. 4, přičemž posílání ovládácích pulsů je třeba přibližně každých 20 ms opakovat. Dále je nutné vzít v potaz, že efektivní pohybový rozsah serva může být omezen v závislosti na jeho roli v rámci robotické ruky. Kontrola dodržení reálného rozsahu však není v kompetenci kontroléru SSC-32, nýbrž musí být implementována na úrovni aplikace, která pro kontrolér generuje příkazy.

Tabulka 1: Zapojení serv do modulu SSC-32.

kloub	číslo kanálu
1	0
2	1, 17
3	2
4	3
5	4



Obrázek 3: Schématické znázornění robotické ruky L6AC. Čísly jsou označeny jednotlivé klouby, tj. místa, v nichž se nacházejí serva.

2.2 Aplikace RoboticArm

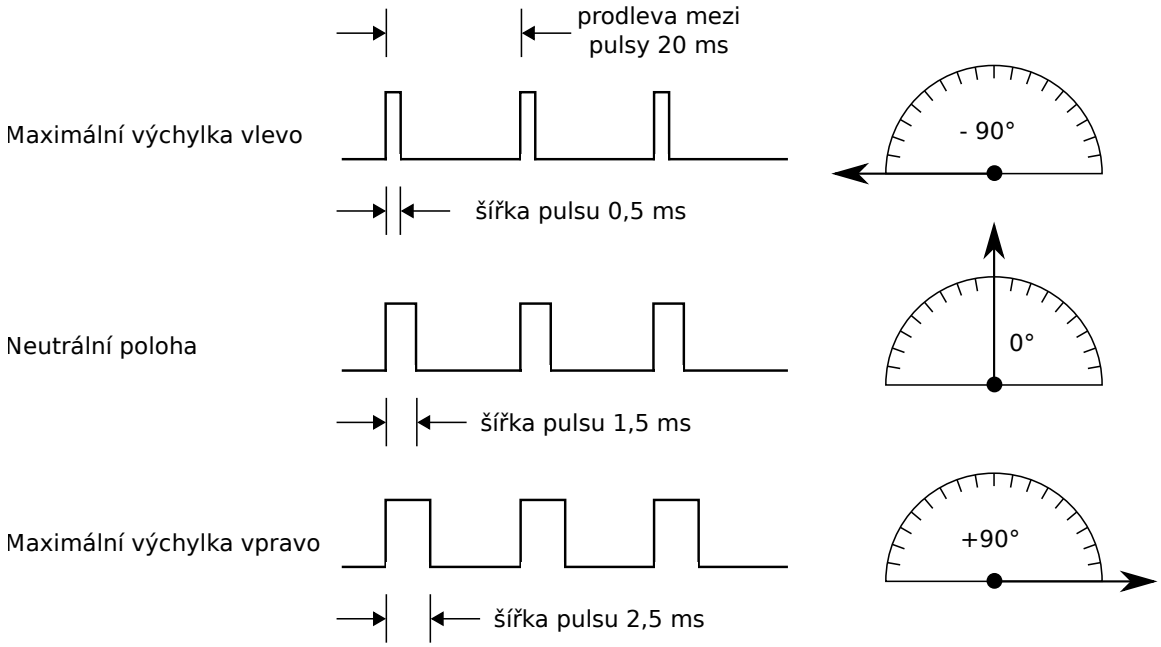
Neméně důležitou součástí celého projektu je i jeho softwarová část. Ta je založena na vlastní linuxové distribuci *i.MX Linux*® [3]. V rámci linuxového systému běží ústřední aplikace RoboticArm, která na jedné straně pomocí grafického rozhraní zprostředkovává interakci s uživatelem, zároveň však generuje i data pro nastavení požadované polohy robotické ruky. Uživatel má k dispozici čtyři režimy aplikace:

Manual První mód, jehož podoba je zachycena na obr. 5a, nabízí pětici posuvníků pro ovládání poloh jednotlivých serv. Již ze samotného názvu lze usoudit, že se *de facto* jedná o ruční režim, v rámci něhož má uživatel naprostou svobodu v ovládání robotické ruky.

Position V tomto módu je uživateli vykreslena hranice dosahu robotické ruky. Kliknutím na bod uvnitř hranice dojde k automatickému přenastavení serv tak, aby se prsty robotické ruky nacházely v místě, které odpovídá souřadnicím kliknutí.

Game Jedná se o zjednodušenou variantu hry „Člověče, nezlob se!“ pro jednoho hráče (viz obr. 5b). Uživatel hází kostkou, přičemž výsledek hodu je zachycen webkamerou a následně zpracován algoritmy pro analýzu obrazu. Podle standardních pravidel hry pak robotická ruka uchopí hrací figurku a přesune ji na novou pozici. V případě problémů s kamerou lze hod kostkou nahradit generátorem náhodných čísel.

Video Poslední režim slouží k přehrání instruktážního videa s ukázkami možností popisované aplikace.



Obrázek 4: Princip nastavení polohy serva pomocí šířky ovládacího pulsu.

Každý požadavek na novou pozici robotické ruky je aplikací nejprve předzpracován, a takto získaný výsledek je následně zaslán ovladači serv. Samotný postup při změně polohy ruky lze shrnout do tří kroků:

1. Na začátku je proveden přepočet cílové polohy na odpovídající šířky ovládacích pulsů pro jednotlivá serva.

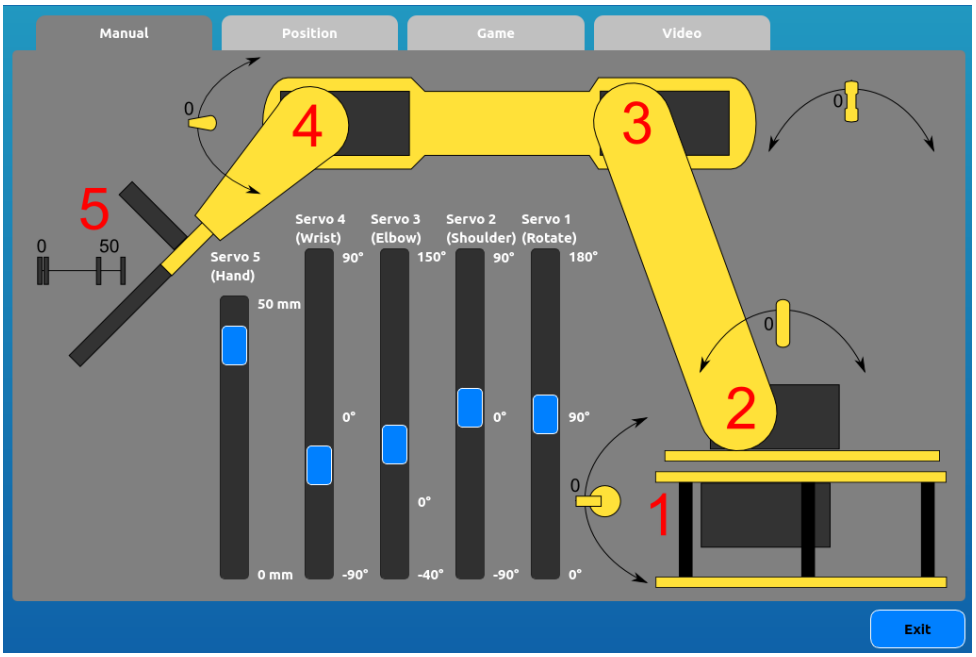
2. Vypočítané parametry jsou poté zapsány do formátu textového řetězce, který nabývá tvaru

   `#`*`id_X`* `P`*`šířka_pulsu_X`* `#`*`id_Y`* `P`*`šířka_pulsu_Y`* *...* `T`*`doba_přesunu`*`\r\n`.

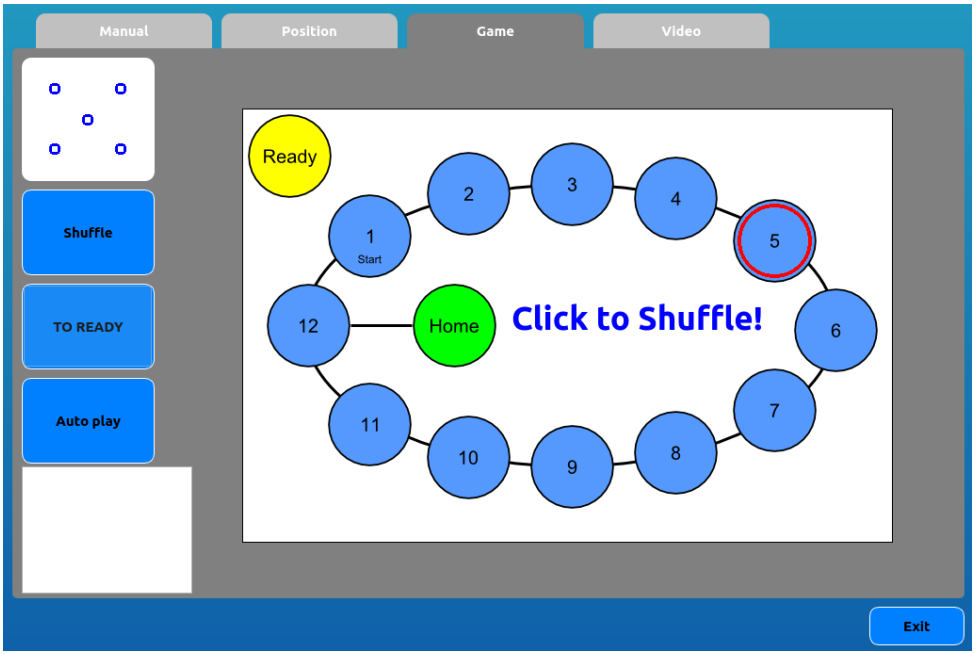
   Identifikátor ***`id_X`*** udává číslo kanálu, v rámci něhož je potřeba nastavit šířku pulsu na hodnotu ***`šířka_pulsu_X`*** v mikrosekundách. Takto lze za sebou řetězit více požadavků, přičemž jako poslední údaj se uvádí atribut ***`doba_přesunu`***, pomocí kterého je specifikován čas v milisekundách, za jak dlouho má být pohyb vyjmenovaných serv dokončen.

3. Výsledný příkaz je následně zapsán do znakového zařízení `/dev/ttyUSB0`, které v linuxovém systému reprezentuje připojený převodník USB/RS232.

K aplikaci lze ještě zmínit skutečnost, že byla napsána objektovým způsobem s využitím programovacího jazyka C++ a frameworku Qt. Pro práci s obrazem se používá knihovna OpenCV, zatímco přehrání instruktážního videa má na starosti knihovna GStreamer. Vzhledem k uvedeným závislostem je pro aplikaci potřeba přibližně 150 MB diskového prostoru.



(a)



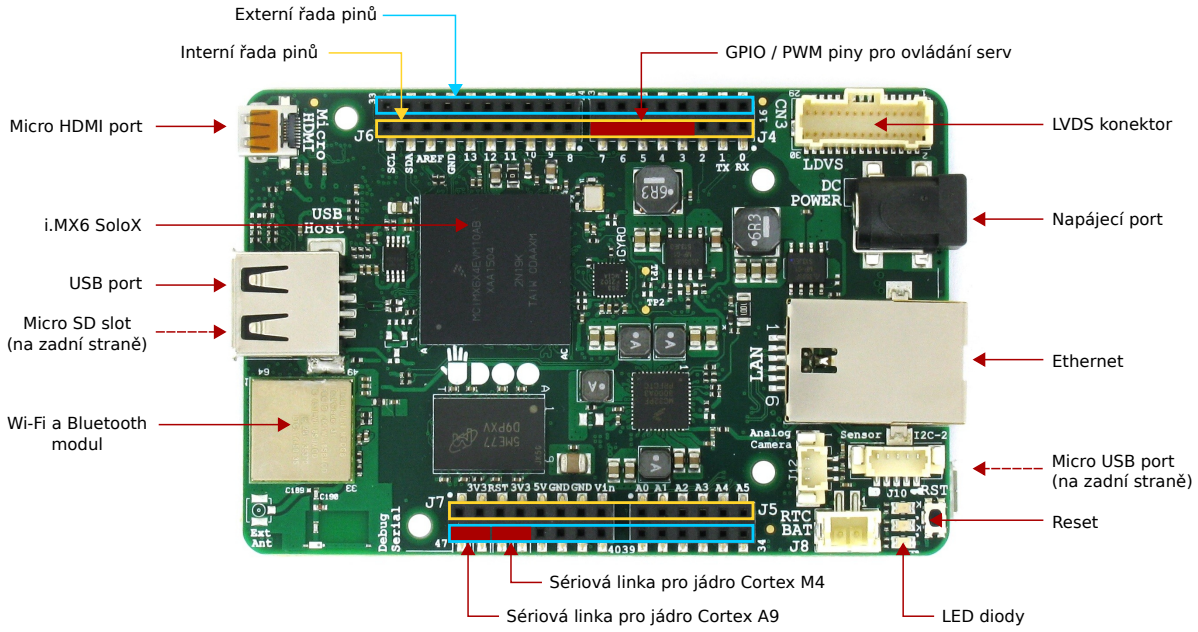
(b)

Obrázek 5: Ukázky aplikace RoboticArm: a) ruční ovládání jednotlivých serv, b) varianta hry „Člověče, nezlob se!“.

3 Popis modifikovaného řešení

Přestože je systém z předchozí kapitoly schopen kontrolovat pohyb robotické ruky, nabízí se několik možností, jak vylepšit jeho funkcionalitu. Začít lze přitom u použité desky i.MX Sabre SD. Ta slouží především k prototypování nových zařízení, a přestože je bohatě osazena různými druhy periferií, chybí na ní piny, které by šlo použít k přímému ovládání serv. Kvůli tomuto omezení proto muselo dojít k použití kontroléru SSC-32. Ale i v případě přímé kontroly serv záhy vyjde na povrch další komplikace: bude-li rozhodnuto implementovat kód pro řízení serv do stávající aplikace RoboticArm, dojde k soupeření s grafickou komponentou o přístup k jedinému výpočetnímu jádru. Vzhledem k použitému operačnímu systému navíc nepůjde zaručit realtime chování aplikace. Součástí platformy i.MX6 SoloX je přitom i druhé výpočetní jádro Cortex M4, které u původního řešení není využito.

Na následujících stranách je proto čtenář seznámen s popisem hardwarových a softwarových komponent vylepšeného řešení, které dokáže ovládát serva bez potřeby dodatečného kontroléru, garantuje realtime chování, a navíc se vyrovná i s případným pádem aplikace RoboticArm. Nejprve dojde k představení nové desky, která disponuje dostatečným počtem pinů, pomocí nichž lze posílat řídící signály pro jednotlivá serva. Samotné generování signálů bude přitom spolu s další programovou logikou prováděno na dedikovaném jádru Cortex M4. Kód běžící na tomto jádru pak bude dostávat povely od GUI aplikace běžící na jádru Cortex A9 prostřednictvím mechanismu meziprocesorové komunikace.



Obrázek 6: Rozmístění komponent na desce UDOO Neo.

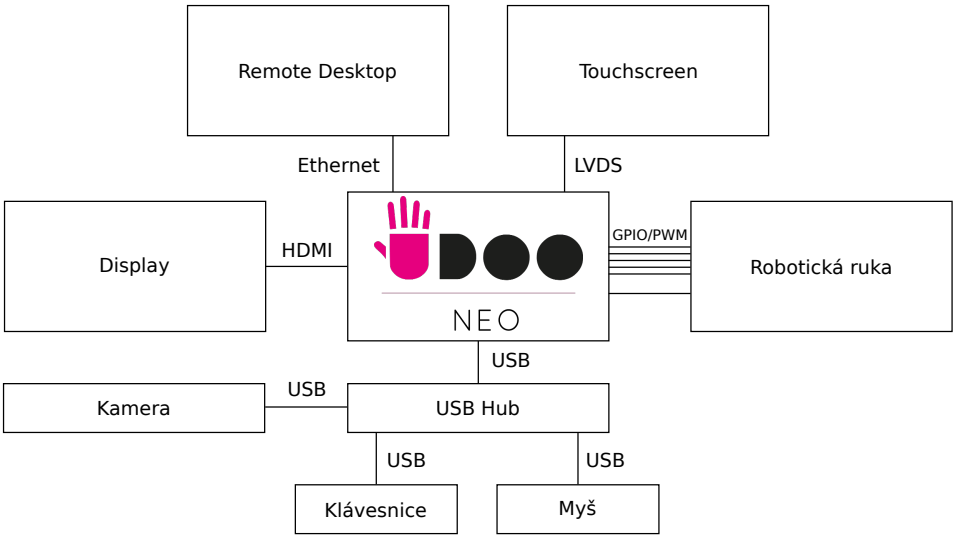
3.1 Hardwarová platforma

Jako náhrada za desku i.MX Sabre SD byla zvolena platforma UDOO Neo [4], a to ve variantě *full*. Jedná se o relativní novinku, které pomohl na trh v roce 2015 úspěšný crowdfunding. Mezi hlavní přednosti zvoleného řešení patří malé rozměry, bohaté možnosti síťové konektivity (ethernet, wifi, bluetooth 4.0), zabudované senzory (akcelerometr, magnetometr, gyroskop), a především široká nabídka pinů pro zprostředkování vstupních či výstupních signálů. Další zajímavostí je i skutečnost, že jednotlivé piny jsou uspořádány do dvou řad, přičemž ve výchozím stavu jsou funkce pinů v rámci vnitřní řady shodné s platformou Arduino UNO. Tímto je zaručena kompatibilita s rozšiřujícími deskami určenými pro uvedenou platformu. K dispozici máme mj. následující rozhraní:

- 8 PWM výstupů,
- 54 GPIO rozhraní,
- 6 analogových vstupů,
- 3 rozhraní I^2C,
- 3 rozhraní UART,
- 2 rozhraní CAN,
- 1 rozhraní SPI.

Je však třeba poznamenat, že některé dvojice či trojice rozhraní sdílí společný pin. Z tohoto důvodu lze v jeden okamžik používat pouze podmnožinu z výše uvedených vstupů a výstupů. Bližší informace o možnostech mapování signálů na jednotlivé piny nalezne čtenář v oficiální dokumentaci [4], příp. schématu [5]. Práce se zaměřuje především na piny s označením 3 – 7 (viz obr. 6), které podporují GPIO i PWM režim, a lze je tak využít pro ovládání serv. Připojením servomotorů přímo do desky UDOO Neo již nebude potřeba ovládací modul SSC-32, čímž se blokové schéma celého systému zjednoduší do podoby na obr. 7.

Srdcem UDOO Neo je stejně jako v případě původní desky i.MX Sabre SD čip i.MX6 SoloX. Jedná se o zástupce šesté generace rodiny mikrokontrolérů i.MX of firmy NXP, jež jsou založeny na architektuře ARM. Samotná zkratka i.MX je odvozena z výrazu „innovative Multimedia eXtension“, z čehož lze snadno usoudit bohaté uplatnění těchto mikrokontrolérů ve spotřební elektronice a automobilovém průmyslu. Značný úspěch architektury ARM v uvedených oblastech souvisí mj. s filozofií integrace veškerých komponent v rámci jednoho čipu. Z tohoto důvodu jsou procesory ARM často označovány termínem SoC – *System on a Chip*. Na stejném čipu se tak kromě výpočetních jader nalézá např. integrovaný řadič pro ethernet, USB či PCIe, ale rovněž i grafický akcelerátor. Tímto se platforma ARM odlišuje od běžnější AMD64, v rámci níž se řadiče periferií obvykle nachází mimo procesor. Mezi další přednosti platformy i.MX (a



Obrázek 7: Blokové schéma navrhovaného zapojení UDOO Neo.

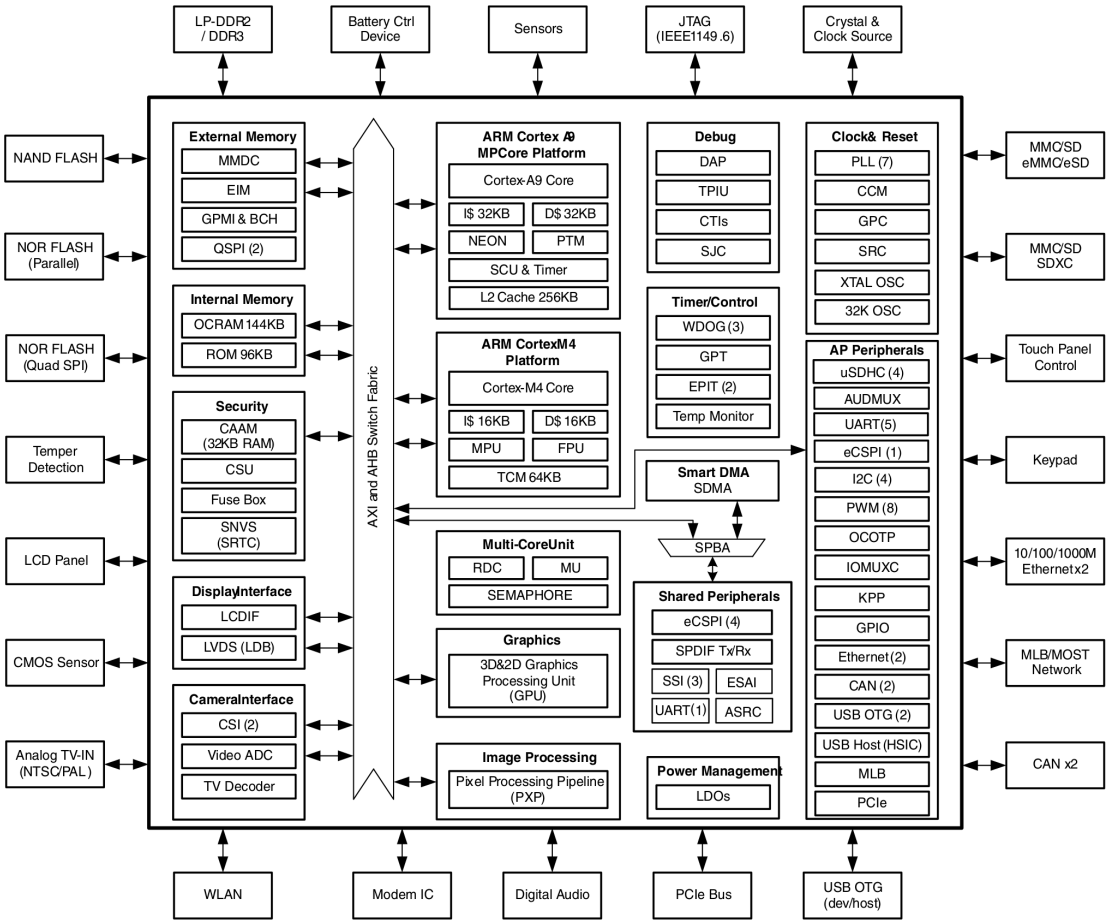
architektury ARM obecně) patří nízká spotřeba, která se v případě varianty SoloX pohybuje v rozmezí 24 mW (tzv. *Deep Sleep Mode*) až 1,4 W (současně běžící benchmarky Dhrystone, CoreMark a 3Dgaming) [6].

Na obr. 8 čtenář nalezne schématické znázornění funkčních bloků čipu i.MX6 SoloX. Kromě universálního výpočetního jádra Cortex A9 a specializovaného jádra Cortex M4 je k dispozici řada podpůrných obvodů pro práci s různými typy pamětí, displeji či periferiemi. Pro případ zpracování 2D a 3D grafiky je navíc k dispozici jádro Vivante GC400T. Jelikož však není cílem práce provádět detailní popis všech součástí, je příp. zájemce odkázán na oficiální dokumentaci [7]. Vzhledem ke zbytku textu je nicméně záhodno představit vybrané komponenty:

ARM Cortex A9 Hlavním výpočetním prostředkem čipu SoloX je 32bitové jádro Cortex A9, které běží s proměnlivou frekvencí až 1 GHz. K dispozici má L1 cache o velikosti 64 kB (32 kB pro data + 32 kB pro instrukce) a 256 kB L2 cache. Součástí jádra je rovněž i koprocesor NEON MPE, jehož SIMD architektura slouží především pro zpracování multimédií a operace s čísly v plovoucí desetinné čárce.

ARM Cortex M4 Součástí platformy SoloX je i druhé 32bitové jádro Cortex M4 s pracovní frekvencí 227 MHz, které se vyznačuje nízkou spotřebou a rychlými odezvami na přerušení [7]. Jádro má k dispozici 32 kB L1 cache (16 kB pro data + 16 kB pro instrukce) a matematický koprocesor, jenž umožňuje operace v jednoduché přesnosti. Inicializaci Cortex M4 je nutné provádět prostřednictvím jádra Cortex A9, které má na starosti nahrání firmwaru a nastavení hodinového taktu.

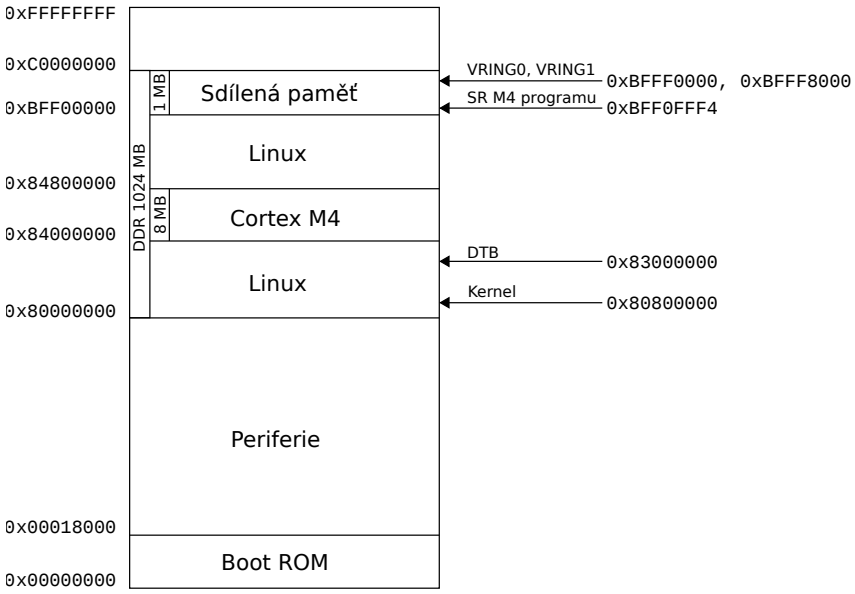
Multi-Core Unit Dříve či později nastane potřeba interakce mezi oběma jádry. K jejímu zajištění slouží modul pro správu jader, který v sobě zahrnuje jednotku *Messaging Unit* (MU) pro meziprocesorovou komunikaci prostřednictvím zasílání zpráv a generování přerušení.



Obrázek 8: Blokové schéma SoC i.MX6 SoloX, převzato z [7].

Neméně důležitou součástí je i komponenta *Resource Domain Controller* (RDC), která má na starosti řízení přístupu ke sdíleným prostředkům. Pomocí tohoto mechanismu je možné mezi jednotlivá jádra mapovat periferie, či přímo celé bloky paměti. S využitím vestavěných hardwarových semaforů lze rovněž zabezpečit přístup k prostředkům, jež jsou sdíleny oběma jádry.

Timer/Control Tato komponenta v sobě zahrnuje jednak prostředky pro monitoring teploty, ale především i tři druhy časovačů, které lze použít v rámci uživatelských aplikací. Prvním z nich je *Watchdog timer* (WDOG), jenž slouží k implementaci tzv. keepalive mechanismu: program musí periodicky obnovovat konfiguraci WDOG časovače, jinak při jeho expiraci dojde k resetu procesoru. Dalším typem časovače je *General Purpose Timer* (GPT), který slouží v případě, kdy je potřeba mít k dispozici informaci o uplynulém čase. Na vstup GPT lze připojit hodinový signál o frekvenci 32 kHz, 24 MHz, příp. je možné použít i externí zdroj signálu. Díky možnosti přeškálovaní je navíc možné docílit i nižších násobků



Obrázek 9: Rozdělení adresního prostoru jádra Cortex A9.

frekvence zvoleného signálu. Aktuální hodnotu časovače lze následně zjistit přečtením dat z dedikovaného registru. Posledním typem časovače je *Enhanced Periodic Interrupt Timer* (EPIT), který primárně slouží k periodickému generování přerušení. Zdroj vstupního hodinového signálu pro EPIT lze zvolit ze stejné množiny jako v případě GPT. Na začátku běhu se nastaví počáteční hodnota EPITu, která se po spuštění časovače začne odpočítávat. Po skončení odpočtu se vygeneruje přerušení a příp. dojde k reinicializaci časovače na počáteční hodnotu.

Jak již bylo naznačeno při popisu vybraných komponent, obě výpočetní jednotky platformy i.MX6 SoloX jsou 32bitové. Z tohoto důvodu lze adresovat maximálně 4GB prostor. Na obr. 9 si může čtenář prohlédnout značně zjednodušené rozdělení adresního prostoru jádra Cortex A9. Dolní polovina rozsahu, tj. adresy `0x00000000` – `0x7FFFFFFF`, jsou využívány především periferiemi, ale nachází se zde i umístění Boot ROM s inicializačním kódem. Horní polovina prostoru, tj. rozsah `0x80000000` – `0xFFFFFFFF`, slouží výhradně k adresaci DDR RAM. Z výše uvedeného čtenář snadno usoudí, že maximální podporovaná velikost operační paměti pro platformu i.MX6 SoloX činí 2 GB. V případě desky UDOO Neo je však k dispozici pouze 1 GB paměti, a tak zůstává rozsah `0xC0000000` – `0xFFFFFFFF` nevyužit.

Přestože by se na téma adresace dalo popsat ještě mnoho stran, pro tuto chvíli se práce zaměří na další aspekty představovaného řešení. Případné zájemce o detailnější rozvržení adresního prostoru je nicméně možné odkázat na 2. kapitolu oficiální dokumentace [7], kde lze rovněž nalézt i adresní schéma jádra Cortex M4. Na obr. 9 však bude ještě poukázáno v dalších kapitolách, v rámci nichž poslouží (nejen) při popisu nahrávání kódu pro jádro Cortex M4.

3.2 Softwarová komponenta pro jádro Cortex A9

Neméně důležitou součástí řešení pro ovládání robotické ruky je i správně fungující operační systém, na kterém poběží aplikace RoboticArm. Tato podkapitola proto začíná výběrem vhodné linuxové distribuce. Následně je popsán způsob meziprocesorové komunikace pomocí protokolu RPMsg. Jedná se o klíčový mechanismus, díky kterému je možné předávat povely pro ovládání serv a příp. zpětně získávat informaci o jejich aktuální nastavení. V poslední části pak dojde ke shrnutí veškerých úprav, jež bylo nutné provést pro získání funkčního řešení.

3.2.1 Linuxová distribuce

Původní řešení používalo vlastní Linuxovou distribuci, která byla vytvořena pomocí nástroje *Yocto*. Nabízí se tak možnost postupovat obdobným způsobem, tj. poskládat jednotlivé funkční vrstvy a následně „upéct“[1] obraz distribuce. Autor práce má však na takto vytvořený systém poněkud kritický názor. Hlavním problémem je především nedořešený mechanismus aktualizací, kdy často jedinou možností je čas od času vytvořit novou distribuci z aktualizovaných zdrojových kódů. Samotné Yocto sice umožňuje pracovat s recepty pro balíčkovací nástroje typu *apt* nebo *yum*, zůstává však problém se správou repozitáře.

Na druhou stranu lze vyjít z již existující distribuce Udoobuntu, jež byla vytvořena autorským týmem desky UDOO Neo. Tato distribuce, k datu psaní práce ve verzi 2.1.2, je založena na portované verzi Lubuntu 14.04 pro platformu ARM. Původní systém byl přitom upraven takovým způsobem, aby fungovaly veškeré komponenty nacházející se na desce. Kvůli tomu Udoobuntu používá vlastní verzi jádra a k dispozici má rovněž i sadu device tree souborů pokrývajících různé režimy desky. Čtenáři neznalému problematiky device tree lze sdělit, že se jedná o unifikovaný mechanismus popisu komponent včetně jejich adresace a konfiguračních parametrů. Podrobnější informace spolu se strukturou těchto souborů nalezne čtenář v knize [8].

K původním repozitářům Lubuntu navíc přibyl i vlastní repozitář s nástroji určenými speciálně pro platformu UDOO. Na tomto místě je vhodné si představit některé z nich:

Web Control Panel Jedná se o webovou aplikaci s aktuálním přehledem o stavu desky, viz obr. 10a. K dispozici jsou informace ohledně modelu desky, jejího výrobního čísla a rovněž lze i vyčíst hodnoty vestavěných senzorů. Vedle toho lze provést základní konfiguraci systému nebo rovnou využít javascriptový emulátor terminálu. Aplikace ve výchozím stavu poslouchá na všech síťových rozhraních s nastavenou IP adresou, na což je třeba brát zřetel při zařazení desky do produkčního prostředí.

Device Tree Editor Nástroj umožňuje snadnou modifikaci systémových device tree struktur. Uživatel je tak schopen z pohodlí grafického rozhraní přiřazovat funkce vybraným pinům, přičemž po uložení změn dojde k automatickému vygenerování aktualizovaných .dts souborů s následnou kompilací do binárního formátu .dtb. Tato aplikace je podobně

[1] Jedná se o *terminus technicus* ze světa nástrojů *Yocto* a *Bitbake*.

jako Web Control Panel založena na webových technologiích (viz též obr. 10b), ve výchozím stavu však není spuštěna.

Arduino IDE Jednou z hlavních předností desky je její kompatibilita s platformou Arduino UNO. Čtenáře tak určitě nepřekvapí přítomnost modifikované verze vývojového prostředí Arduino IDE, do nějž byla přidána podpora pro UDOO Neo. Při práci s deskou lze tudíž používat i *sketch*[2] formát zdrojového kódu známého ze světa Arduina.

Přestože je možnost použití Arduino IDE pro mnoho uživatelů jistě zajímavá, z hlediska diplomové práce je důležitější se přesunout k popisu mechanismu, jak se ze zdrojových souborů pro Arduino stane běžící program. Tato znalost bude později využita při nasazení vlastního programu pro ovládání serv. Celou proceduru je možné shrnout do následujících kroků:

1. Původní zdrojové kódy pro Arduino sketch jsou doplněny o makra a definice funkcí do podoby zdrojového kódu v jazyce C.

2. Takto vzniklý kód je pomocí překladače `armgcc` zkompilován a následně spolu s knihovnou realtime operačního systému *MQX* [9] slinkován do binárního formátu *elf*.

3. Pomocí nástroje `objcopy` se z binární aplikace odstraní nepotřebná metadata, čímž se program převede do binárního formátu *bin*.

4. Získaný binární soubor se pomocí skriptu `udooneo-m4uploader` a nástroje `mqx_upload_on-_m4SoloX` zkopíruje na specifické místo v operační paměti. Poté dojde k resetu jádra Cortex M4, které následně začne vykonávat instrukce nahraného programu.

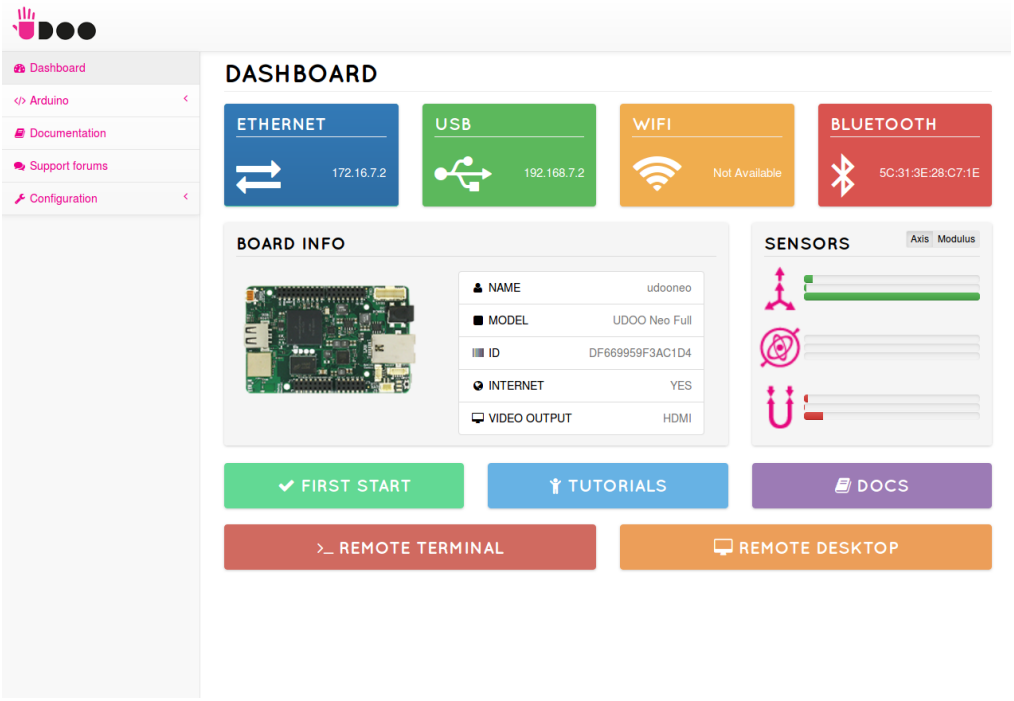
5. Do souboru `/var/opt/m4/m4last.fw` se vytvoří kopie binárního souboru kvůli zajištění perzistence programu při restartu desky.

Uživatelský kód pro Arduino tudíž běží jakožto úloha v rámci realtime operačního systému MQX od firmy Freescale (NXP). Na tomto místě je však třeba podotknout, že vývoj systému MQX v současnosti spíše stagnuje[3]. Minimálně v oblasti bezplatných řešení jej navíc pomalu, ale jistě nahrazují jeho konkurenti, především pak realtime systém FreeRTOS [10]. Autorovi se bohužel nepodařilo zjistit, co přesně vedlo tvůrce UDOO Neo k volbě systému MQX. Jednou z hypotéz však může být, že v době vydání desky ještě nebyla dokončena portace konkurenčního FreeRTOSu do formy BSP balíku [11] pro platformu i.MX6 SoloX.

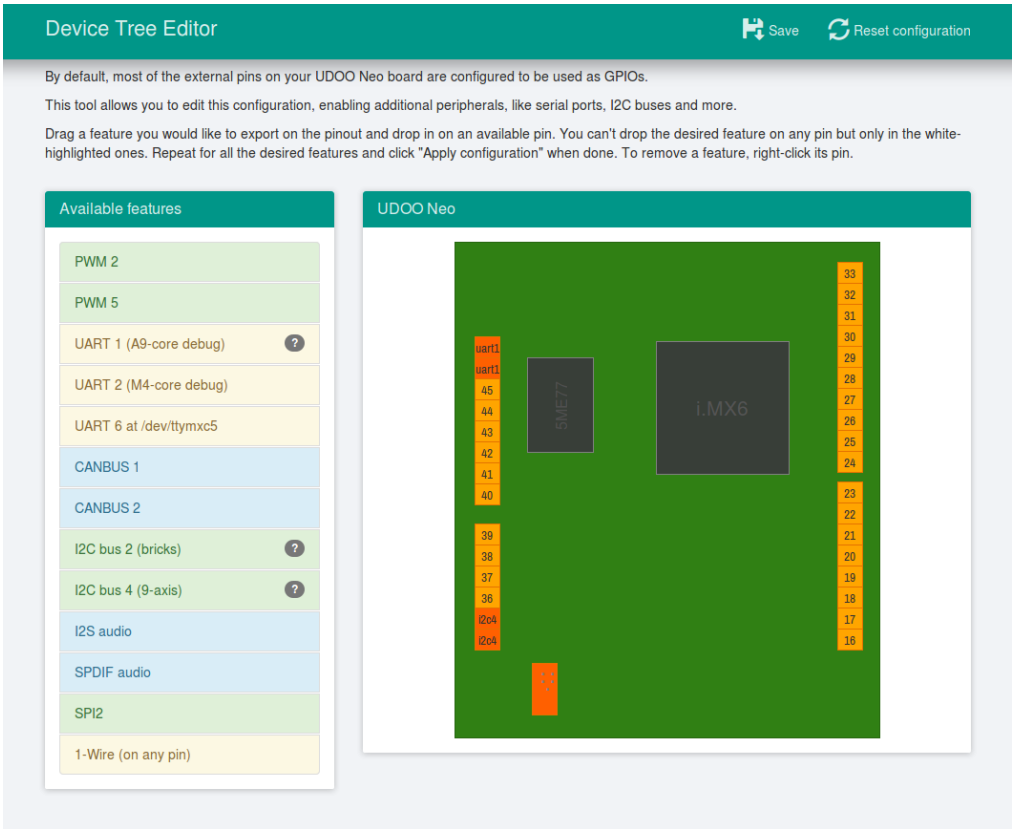
Při popisu zpracování kódu pro Arduino bylo zmíněno specifické místo pro nahrání programu. V případě použité platformy se jedná o adresu `0x8400000` – čtenář je tímto odkázán na dřívější obr. 9. Uvedenou adresu nelze jednoduše změnit, neboť je specifikována na mnoha místech. Mezi ně patří např. sekce v používaných device tree souborech, viz výpis 1.

[2] `https://www.arduino.cc/en/tutorial/sketch`

[3] Zatím poslední verze MXQ 4.1 vyšla na začátku roku 2014. Společnost NXP nicméně vedle toho nabízí i modifikovaný systém MQX v5, který je ale zpoplatněn.



(a)



(b)

Obrázek 10: Grafická rozhraní vybraných aplikací pro Udoobuntu: a) Web Control Panel, b) Device Tree Editor.

Výpis 1: Segment dekompilovaného souboru `/boot/dts/imx6sx-udoo-neo-full-hdmi-m4.dtb`.

```
...
m4_memory: m4@0x84000000 {
   no-map;
   reg = <0x84000000 0x00800000>;
};
...
```

Adresa sloužící k nahrání kódu pro jádro Cortex M4 se rovněž používá v rámci konfiguračních proměnných zavaděče U-Boot [12], který je součástí obrazu s distribucí Udoobuntu. Vzhledem ke skutečnosti, že se spolu s Udoobuntu dodává upravená verze tohoto zavaděče, je záhodno popsat mechanismus bootování s důrazem na provedené změny:

1. Po připojení napájení dojde nejprve k inicializaci jádra Cortex A9, jež vykoná instrukce uložené v bootovací ROM.

2. Jelikož jádro Cortex A9 očekává v případě desky UDOO Neo umístění zavaděče OS na SD kartě, dojde k přečtení předem známé oblasti, kterou je začátek 2. kB. Zde je umístěna první fáze zavaděče, tzv. *Secondary Program Loader* (SPL). Ta má na starosti inicializaci vybraných registrů SoC, nastavení časování paměti a načtení druhé fáze zavaděče, která se nachází na 70. kB SD karty.

3. Druhá fáze zavaděče již má k dispozici přístup k souborovému systému na kartě. Díky tomu dokáže zavaděč přečíst obsah souboru `/boot/uEnv.txt`. V něm jsou uloženy specifické volby pro UDOO Neo, mezi něž patří konfigurace výstupu videa (HDMI nebo LVDS), požadavek na start jádra Cortex M4 a příp. zdali došlo k úpravě device tree pomocí Device Tree Editoru. Na základě uvedených informací se ze souborového systému vybere správná varianta binárního device tree (DTB) souboru.

4. Rovněž se prozkoumá, zdali je k dispozici soubor `/var/opt/m4/m4last.fw`. Pokud ano, zkopíruje se jeho obsah na adresu `0x84000000` a provede se inicializace jádra Cortex M4.

5. V posledním kroku se na předem dané adresy v paměti načte soubor `/boot/zImage` s linuxovým jádrem a zvolený DTB soubor, viz též obr. 9. Následně je linuxovému jádru předána kontrola nad systémem.

Výběr DTB souboru na základě informací uvedených v `/boot/uEnv.txt` má však jedno nepříjemné omezení: modifikovaný kód U-Bootu předpokládá specifické pojmenování DTB souborů, jež reflektuje konfigurační volby. Název souboru s device tree tak musí být pojmenován ve tvaru `imx6sx-udoo-full-{hdmi|lvds}[-m4].dtb`. Uvedené omezení bude nutné vzít v potaz, dojde-li k rozhodnutí používat vlastní DTB soubor.

Pro úplnost je čtenáři v rámci výpisu 2 poskytnut výstup ze sériové konzole, který zachycuje bootování platformy počínaje třetím krokem výše zmíněného postupu až do startu linuxového jádra.

Výpis 2: Boot platformy UDOO Neo.

```
switch to partitions #0, OK
mmc0 is current device
reading uEnv.txt
163 bytes read in 10 ms (15.6 KiB/s)
Running bootscript from mmc ...
Device Tree: dts/imx6sx-udoo-neo-full-hdmi-m4.dtb
reading /zImage
4246328 bytes read in 226 ms (17.9 MiB/s)
Booting from mmc ...
reading dts/imx6sx-udoo-neo-full-hdmi-m4.dtb
46092 bytes read in 37 ms (1.2 MiB/s)
Kernel image @ 0x80800000 [ 0x000000 - 0x40cb38 ]
## Flattened Device Tree blob at 83000000
   Booting using the fdt blob at 0x83000000
   Using Device Tree in place at 83000000, end 8300e40b
Switched to ldo_bypass mode!

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 3.14.56-02051-gaff97f7 PREEMPT Sun Feb 26 15:46:34 CET 2017
[    0.000000] CPU: ARMv7 Processor [412fc09a] revision 10 (ARMv7), cr=10c53c7d
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
[    0.000000] Machine model: UDOO Neo Full
[    0.000000] Reserved memory: reserved region for node 'm4@0x84000000': base 0x84000000, size
    8 MiB
...
```

Ve výpisu stojí především za zmínku poslední řádek s rezervací 8MB oblasti, jež byla specifikována pomocí záznamu v DTB souboru. Zmíněný úsek paměti poslouží k uchování běžícího programu pro jádro Cortex M4.

Případné zájemce o další informace k distribuci Udoobuntu lze odkázat na oficiální dokumentaci [4]. Text se nyní bude věnovat metodám meziprocesorové komunikace, které umožní výměnu informací o poloze serv v rámci obou výpočetních jader.

3.2.2 Mechanismus meziprocesorové komunikace

V průběhu let došlo k implementaci mnoha řešení komunikace mezi více procesory. Většina z nich však předpokládá symetrický multiprocessing (např. OpenMP), při kterém kterém je použita identická architektura CPU, příp. je nasazen společný operační systém. V případě omezení se na asymetrický multiprocessing lze v zásadě uvažovat o dvou mechanismech, a to MCC a RPMsg:

MCC Knihovna *MultiCore Communication* [13] funguje jako podsystém realtime operačního systému MQX. Kromě toho je k dispozici i její portace do upravené verze linuxového jádra od společnosti NXP. Knihovna ke své činnosti vyžaduje sdílenou paměť, systém přerušení

a sadu hardwarových semaforů, jimiž se řídí přístup do sdílené paměti. Jednotlivé koncové body jsou adresovány trojicí hodnot (`core`, `node`, `port`). Parametr `core` slouží k identifikaci jádra v rámci SoC (pro i.MX6 SoloX hodnota 0 označuje Cortex A9 a 1 Cortex M4). Hodnota `node` v případě Linuxu odkazuje na běžící proces, zatímco u MQX tento parametr nemá žádný význam a je vždy nulový. Poslední identifikátor `port` je uživatelsky nastavitelný a slouží k oddělení současně probíhajících konverzací. Data se mezi koncovými uzly zasílají na základě alokace volných bufferů o fixní velikosti. Všechny koncové uzly přitom sdílí společnou množinu volných bufferů, které se používají pro oba směry komunikace. Uvedený mechanismus je v systému Udoobuntu využíván kvůli emulaci sériové linky, jež slouží pro komunikaci s programem běžícím v rámci MQX.

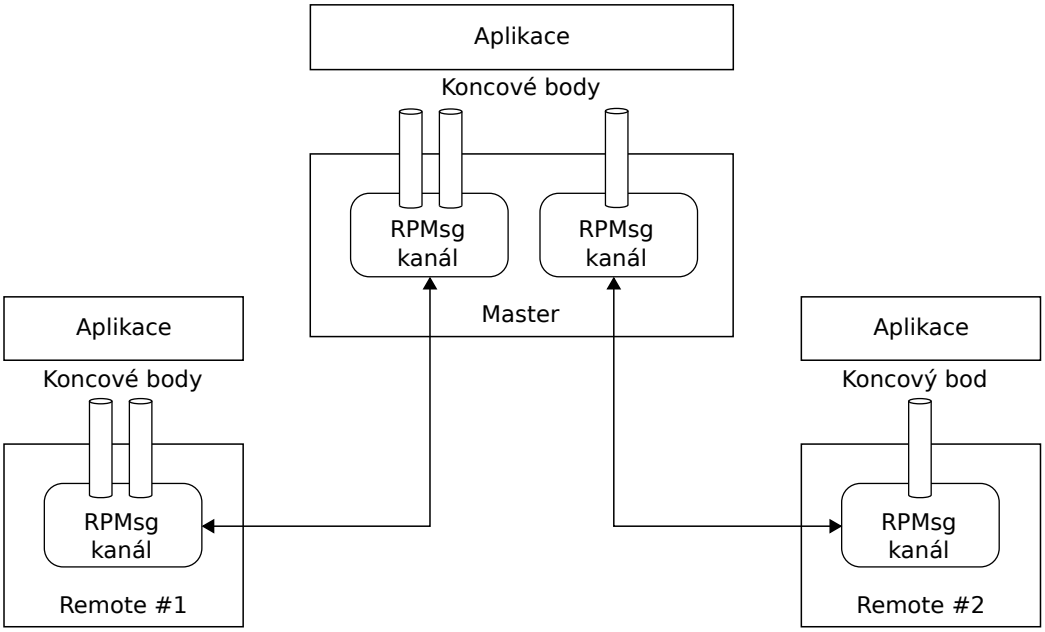
RPMsg *Remote Processor Messaging* je komponenta frameworku OpenAMP [14] pro zasílání zpráv v rámci heterogenních víceprocesorových systémů. Z hlediska požadavků na platformu postačuje pouze sdílená paměť a systém přerušení. Ve srovnání s MCC tudíž není potřeba hardwarových semaforů. Uvedené skutečnosti bylo docíleno tím, že je podstatná část RPMsg funkcionality založena na architektuře *virtio* [15]. Komponenta RPMsg je k dispozici pro jádro Cortex M4 ve formě BSP balíku realtime operačního systému FreeRTOS [16]. V případě linuxového jádra lze základní podporu pro RPMsg vystopovat již u vanilla verze 3.4.

Ve zkratce tak lze říci, že mechanismus MCC je přirozenou volbou pro realtime systém MQX, zatímco v případě systému FreeRTOS je vhodné zvolit RPMsg. Jelikož se náplň práce ubírá směrem k FreeRTOSu[4], jsou další odstavce zaměřeny na bližší popis RPMsg.

Mechanismus komunikace pomocí RPMsg se skládá z několika funkčních komponent, viz obr. 11. Mezi dvojicí jader je vytvořen RPMsg kanál, který je reprezentován textovým názvem a číselnými identifikátory obou konců. Jednomu z jader v rámci kanálu je staticky nastavena role Master, zatímco druhé jádro má nastavenu roli Remote. Úkolem Mastera je především inicializace a správa datových struktur ve sdílené paměti, které se používají pro meziprocesorovou komunikaci. Z tohoto důvodu bývá jako Master obvykle zvolen primární procesor (v případě i.MX6 SoloX je jím jádro Cortex A9), ale není to bezpodmínečně nutné.

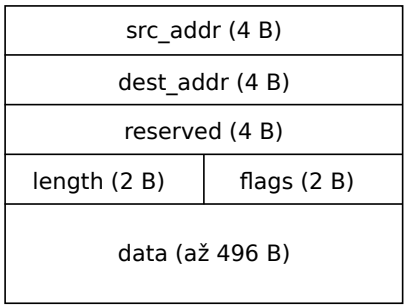
Jelikož se v rámci jednoho kanálu vykytují právě dvě jádra, jedná se o tzv. point-to-point topologii. Samotný kanál však může sloužit k několika nezávislým datovým tokům. K jejich rozlišení slouží koncept tzv. koncový bodů. Každý koncový bod je v rámci kanálu jednoznačně identifikován 32bitovou adresou. Chce-li aplikace přijímat data, zaregistruje si koncový bod se zvolenou adresou a callback funkcí. Veškeré příchozí zprávy jsou pak na základě cílové adresy směrovány do odpovídajících callback funkcí. V rámci každého RPMsg kanálu je přitom na obou koncích implicitně vytvořen jeden koncový bod. Používají-li komunikující aplikace pouze jeden datový tok, nemusí se konceptem koncových bodů vůbec zabývat.

[4] Přinejmenším kvůli explicitnímu požadavku na tento systém v zadání práce.



Obrázek 11: Schématické znázornění RPMsg komponent, převzato z [14].

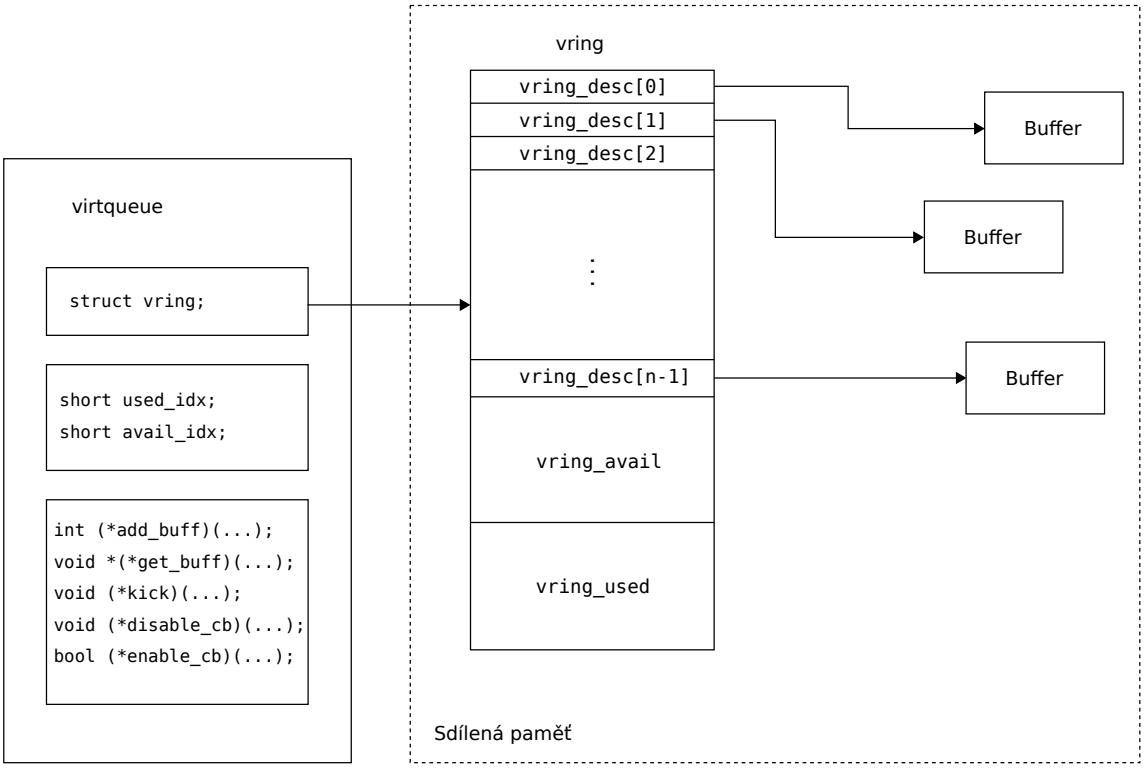
Aplikační data se v rámci RPMsg posílají formou paketů o maximální velikosti 512 B, přičemž jejich hlavička má fixní délku 16 B, viz obr. 12. Samotná hlavička se skládá z pěti položek: položka `src_addr` (4 B) je určena k identifikaci zdrojového koncového bodu, zatímco položka `dest_addr` (rovněž 4 B) obsahuje adresu cílového koncového bodu. Položka `reserved` slouží k zarovnání hlavičky na 16 B. Poslední dvě položky v hlavičce paketu obsahují délku uživatelských dat (`length`, 2 B) a RPMsg příznaky (`flags`, rovněž 2 B). V době psaní textu však nebyl žádnému příznaku přidělen význam. Pro data uvnitř paketu zbývá 496 B místa.



Obrázek 12: Formát RPMsg zprávy.

3.2.3 Komunikační framework virtio

Přenos RPMsg zpráv je implementován prostřednictvím frameworku virtio, jenž byl původně vyvinut pro paravirtualizaci ovladačů v rámci hypervizoru KVM [15]. Samotné virtio poskytuje jednotný mechanismus pro transport dat a API, díky němuž lze nad tímto frameworkem



Obrázek 13: Znázornění datových struktur `virtqueue` a `vring`.

psát vlastní ovladače. Tak je tomu i v případě RPMsg, kdy pro linux existuje několik jaderných modulů implementujících RPMsg funkcionalitu. Tyto moduly slouží jednak jako jeden z konců komunikačního kanálu, zároveň ale také zprostředkovávají přístup k RPMsg procesům v uživatelském prostoru, např. pomocí emulace znakového zařízení `/dev/ttyRPMSG`. V případě realtime systému FreeRTOS je k dispozici knihovna [16] umožňující vytvořit vlastní zakončení RPMsg kanálu. Zároveň jsou poskytnuty funkce pro správu komunikačních bodů a příjem, příp. odesílání RPMsg zpráv.

Transport dat je v rámci virtio řešen pomocí rozhraní `virtqueue`, viz obr. 13. Jedná se o reprezentaci fronty, která obstarává jeden směr komunikace a zároveň poskytuje abstrakci pro práci se strukturou `vring`. Před použitím rozhraní `virtqueue` je potřeba, aby byly v rámci ovladače zařízení implementovány veškeré operace pro práci s frontou. Jedná se především o funkce:

- `add_buff`: pro přidání nového bufferu do fronty,
- `get_buff`: slouží k odebrání bufferu z fronty,
- `kick`: notifikace při změně stavu fronty,
- `disable_cb` / `enable_cb`: vypnutí, příp. zapnutí notifikací při změně stavu fronty.

Zasílaná data jsou uchována uvnitř struktury `vring`. Tato struktura se fyzicky nachází ve sdílené paměti, díky čemuž k ní mohou přistupovat obě komunikující strany. Je složena ze tří komponent:

Tabulka deskriptorů Obsahuje 64bitovou adresu bufferu pro aplikační data ve sdílené paměti, 32bitový atribut udávající velikost bufferu a mj. i index dalšího deskriptoru kvůli možnosti zřetězení bufferů. Použití 64bitové adresy bylo zvoleno z důvodu zajištění jednotného formátu deskriptoru na všech platformách.

Kruhový buffer AVAIL Obsahuje indexy řádků v tabulce deskriptorů, jež jsou k dispozici pro použití.

Kruhový buffer USED Obsahuje indexy využitých řádků v tabulce deskriptorů.

Celkový proces komunikace v obou směrech si lze prohlédnout na obr. 14. Zároveň je vhodné zdůraznit, že vzhledem k jednosměrné povaze mechanismu virtio je pro každý směr potřeba samostatná fronta `virtqueue` spolu se strukturou `vring`. V případě zasílání zprávy ve směru Master → Remote probíhá přenos následujícím způsobem:

1. Master nejprve provede rezervaci bufferů, jež poslouží k odeslání dat. Z kruhového bufferu USED proto odebere indexy do tabulky deskriptorů `vring_desc`.

2. Do získaných bufferů se zapíše RPMsg hlavička a aplikační data.

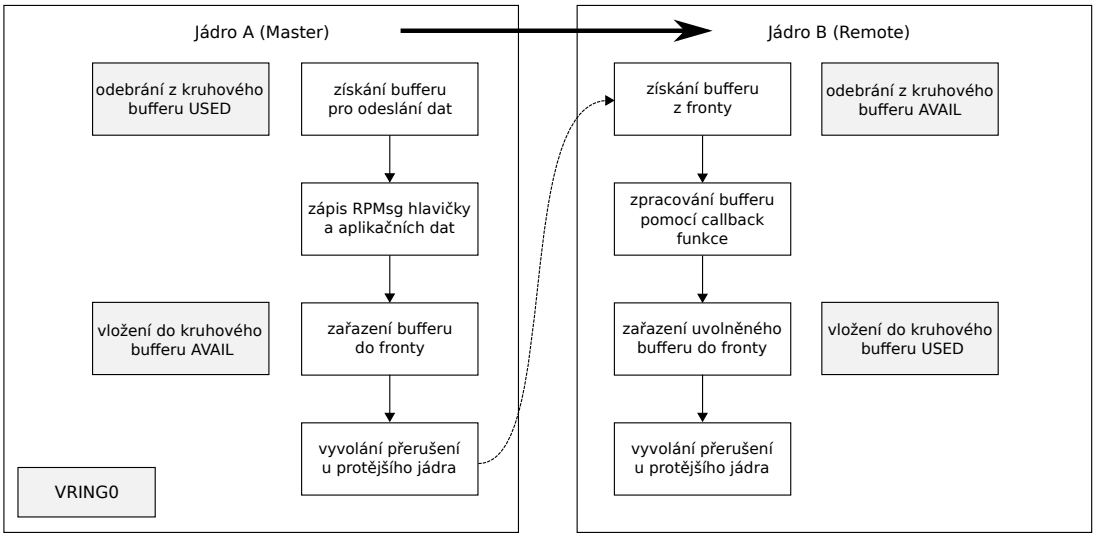
3. Index prvního použitého záznamu v tabulce deskriptorů se vloží do kruhového bufferu AVAIL. Vzhledem k možnosti zřetězení záznamů z tabulky deskriptorů není třeba vkládat zbývající indexy.

4. Následně dochází k notifikaci vzdáleného konce, a to formou *meziprocesorového přerušení* (IPI). Uzná-li však příjemce za vhodné, může požádat odesílatele o neprovádění notifikací; v takovém případě pak musí příjemce periodicky kontrolovat stav fronty.

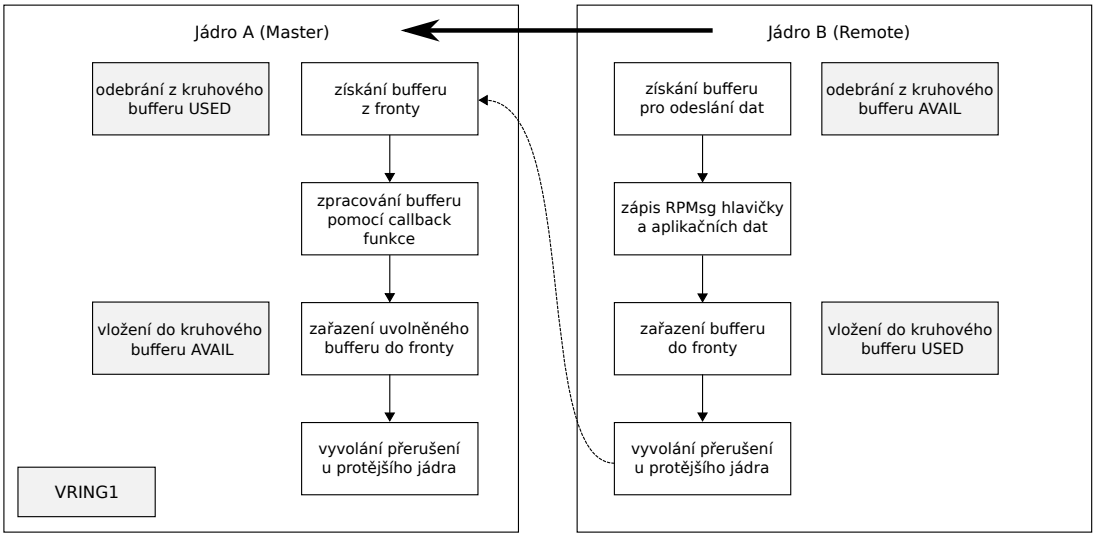
5. Vzdálený konec Remote odebere z bufferu AVAIL index ukazující do tabulky deskriptorů. Tímto získá přístup k zaslanému paketu, u nějž přečte RPMsg hlavičku a na základě adresy cílového koncového bodu přepošle aplikační data odpovídající callback funkci.

6. Po zpracování dat jsou uvolněné indexy zařazeny do kruhového bufferu USED a původní odesílatel je informován pomocí IPI, že již došlo ke zpracování přijaté zprávy.

Po důkladném studiu obr. 14 si čtenář zajisté všiml jisté asymetrie v rámci posílání RPMsg zpráv opačným směrem: posílá-li Remote zprávu uzlu Master, role kruhových bufferů USED a AVAIL jsou obráceny (viz obr. 14b). Je to kvůli tomu, že bez ohledu na směr komunikace má Master na starosti správu bufferů – je jejich poskytovatelem. To mu *de facto* umožňuje kontrolovat objem komunikace, který se zasílá v rámci virtio.



(a) Master → Remote



(b) Remote → Master

Obrázek 14: Princip komunikace pomocí virtio pro jednotlivé směry, převzato z [14]. Šedou barvou jsou vyznačeny odpovídající operace nad strukturami `VRING0` (směr Master → Remote) a `VRING1` (směr Remote → Master).

Právě představený mechanismus meziprocesorové komunikace má však v případě desky UDOO Neo zásadní problém: u linuxového jádra dodávaného v rámci distribuce Udoobuntu chybí několik důležitých komponent, což v důsledku znemožňuje nasazení RPMsg. O způsobu, jak se s tímto problémem vypořádat, se čtenář dozví v následující kapitole.

3.2.4 Úpravy softwaru

Jak již bylo poznamenáno dříve, použití existující linuxové distribuce přináší oproti vlastnoručně sestavenému systému mnoho nesporných výhod. Při nasazení systému Udoobuntu v rámci projektu pro ovládání robotické ruky však bylo nutné provést několik modifikací. Jedna z největších změn se týká samotného linuxového jádra, neboť dodávaná verze nepodporuje komunikaci pomocí mechanismu RPMsg. V případě zbývajících úprav se však již jedná pouze o drobné zásahy, a celkový proces modifikace tudíž ve výsledku není výrazně komplikovaný.

Linuxové jádro

Distribuce Udoobuntu 2.1.2 využívá linuxové jádro verze `3.14-1.0.x-udoo`, jež dle oficiální dokumentace [4] vychází z jádra 3.14.56 od společnosti Freescale (NXP). Vzhledem k chybějící podpoře RPMsg však bylo potřeba uvedené jádro nahradit. Původně se uvažovalo o nasazení některé z variant linuxového jádra od společnosti Freescale určených pro platformu i.MX6. Žádné z těchto jader však není optimalizováno pro desku UDOO Neo, což obnáší nefunkčnost některých komponent, především HDMI výstupu. Autor práce byl proto nucen vytvořit patch pro distribuční verzi jádra, s jehož pomocí lze RPMsg funkcionalitu zprovoznit. Jelikož se však nepovedlo dohledat původní verzi jádra uvedenou v dokumentaci UDOO, stalo se výchozím bodem jádro `imx_3.14.52_1.1.0_ga`[5], rovněž od společnosti Freescale.

U předlohy bylo nejprve třeba izolovat kód související s RPMsg, a následně provést portaci do distribučního jádra `3.14-1.0.x-udoo`. V tab. 2 čtenář nalezne přehled souborů spolu s typem úpravy, které bylo nutné v rámci implementace RPMsg provést. Celkem bylo pozměněno 12 souborů, přičemž u některých z nich muselo dojít k podrobnější analýze důsledků provedených změn. Výsledný patch je dán k dispozici na elektronickém médiu, jež je součástí práce. Po jeho aplikaci na zdrojové soubory distribučního jádra je třeba takto upravené jádro zkompilovat a přenést do souborového systému Udoobuntu. V příloze B.1 je dán k dispozici detailní návod, jak docílit modifikované verze jádra.

S problematikou linuxového jádra do značné míry souvisí i správná konfigurace device tree. V rámci zprovoznění meziprocesorové komunikace bylo potřeba vložit odpovídající RPMsg sekci do DTS souborů:

```
...
&rpmsg{
       compatible = "fsl,imx6sx-rpmsg";
       status = "okay";
};
...
```

[5] `http://git.freescale.com/git/cgit.cgi/imx/linux-imx.git/?h=imx_3.14.52_1.1.0_ga`

Tabulka 2: Seznam modifikovaných souborů linuxového jádra.

Název souboru	Typ úpravy
drivers/rpmsg/	
Kconfig	ruční sloučení
Makefile	vložení nových řádků
imx_rpmsg_pingpong.c	nový soubor
imx_rpmsg_tty.c	nový soubor
include/linux/	
imx_rpmsg.h	nový soubor
arch/arm/mach-imx/	
Kconfig	ruční sloučení
Makefile	vložení nových řádků
imx_rpmsg.c	nový soubor
mu.c	ruční sloučení
arch/arm/boot/dts/	
imx6sx-sdb-m4.dts	vložení nových řádků
imx6sx.dtsi	vložení nových řádků
imx6sx-udoo-neo-m4.dtsi	ruční sloučení

Kromě toho došlo k odstranění zařízení `/dev/ttyMCC`, jež sloužilo v rámci původně použitého komunikačního mechanismu MCC:

```
...
&mcctty{
        status = "disabled";
};
...
```

Právě představené úpravy DTS souborů jsou již součástí patche jádra. Rovněž je vhodné podotknout, že provedené zásahy v device tree nemají žádný negativní dopad na funkčnost uživatelské aplikace Device Tree Editor a rovněž nejsou ani tímto programem ovlivňovány.

Konfigurace systému

Mezi další nutné změny patří především nahrání jaderného modulu `imx_rpmsg_tty`, který zprostředkovává přístup k RPMsg na základě emulace `tty` rozhraní pomocí znakového zařízení `/dev/ttyRPMSG`. Ve výchozím stavu však uvedené zařízení patří uživateli root. Jelikož aplikace RoboticArm běží pod neprivilegovaným uživatelem, je nutné nastavit pravidla manažeru *udev* tak, aby mělo zařízení nastavenu správnou vlastnickou skupinu. V neposlední řadě je potřeba zajistit automatické nahrávání modulu při startu systému. Toho je docíleno úpravou souboru `/etc/modules`. Detailní návod, jak dosáhnout uvedených změn, nalezne čtenář v sekci B.2 přílohy.

Aplikace RoboticArm

V případě převzaté varianty [17] GUI aplikace RoboticArm je situace snadná. První ze dvou potřebných změn je nastavení proměnné *mInterface* (soubor `roboticarm.cpp`). Ta specifikuje cestu ke znakovému zařízení, jež v systému reprezentuje kontrolér SSC-32 pro ovládání serv. Jelikož práce směřuje k řízení serv pomocí jádra Cortex M4, je potřeba hodnotu proměnné nastavit na RPMsg rozhraní `/dev/ttyRPMSG`. Druhá změna spočívá ve změně identifikátoru připojené webkamery. V konstruktoru třídy *Camera* (soubor `camera.cpp`) se při inicializaci objektu *cv::VideoCapture()* počítá s fixní hodnotou 0. Jelikož však UDOO Neo disponuje 8bitovým paralelním rozhraním pro kameru, dochází v případě webkamery připojené přes USB rozhraní k posunu indexu na 1. Proto je třeba inicializovat *cv::VideoCapture()* právě hodnotou 1.

Popsané změny není třeba ve zdrojovém kódu aplikace provádět, neboť verze programu, která je dodávána na přiloženém elektronickém médiu, má již všechny úpravy implementovány. Aplikaci tudíž stačí pouze zkompilovat a spustit. Čtenář je odkázán na část B.3 s podrobným návodem.

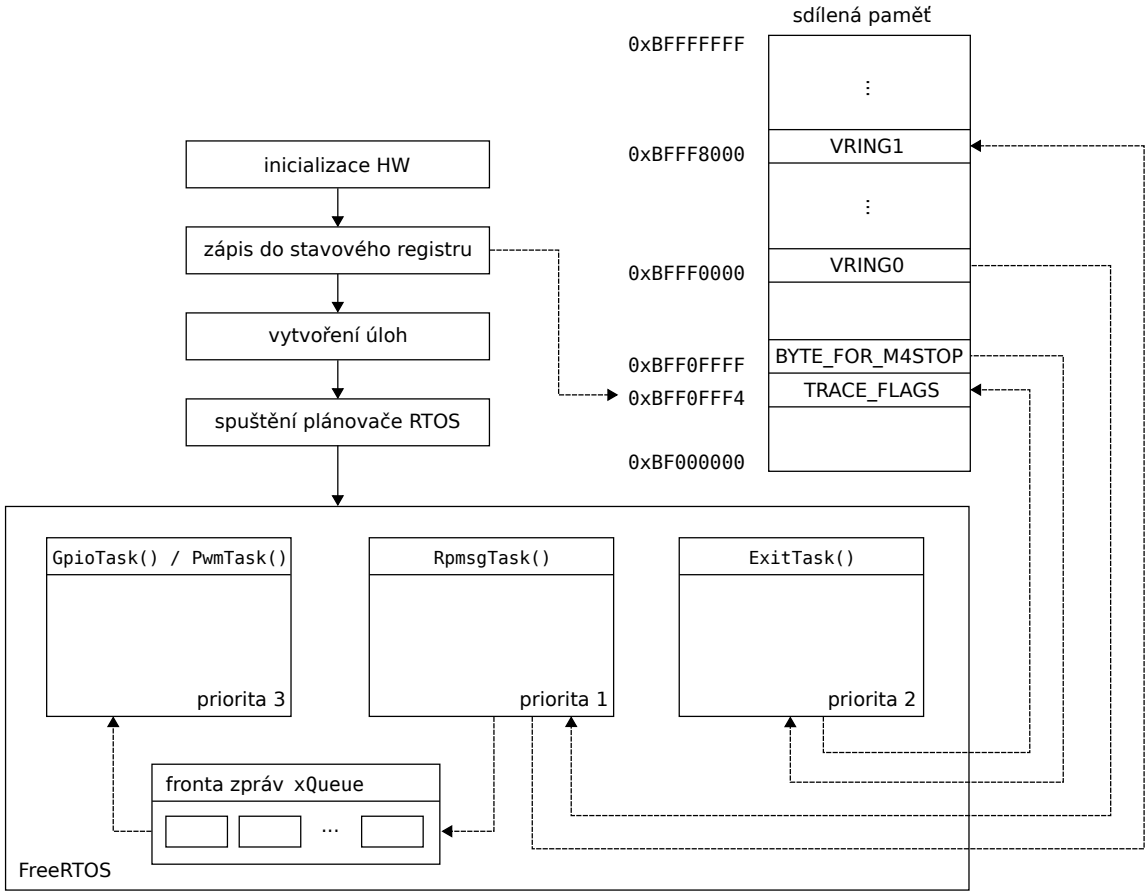
3.3 Softwarová komponenta pro jádro Cortex M4

Jeden z hlavních úkolů diplomové práce spočívá v implementaci ovladače serv, který by dokázal nahradit nezávislý modul SSC-32. Následující kapitola proto čtenáře seznamuje s popisen navrženého softwaru, jenž je založen na realtime operačním systému FreeRTOS. Představeny jsou rovnou dvě varianty programu, které se liší způsobem generování výstupních pulsů. V prvním případě se používá periodické zapínání a vypínání GPIO pinů, přičemž doba zapnutého stavu je určena pomocí dedikovaného časovače EPIT. Druhá, vylepšená varianta ke své činnosti využívá PWM rozhraní.

3.3.1 FreeRTOS

V minulé kapitole bylo provedeno srovnání dvou operačních systémů reálného času, a to MQX a FreeRTOS. Přestože již byly uvedeny argumenty vedoucí k upřednostnění systému FreeRTOS, nabízí se otázka, zdali je při psaní kódu ovladače serv skutečně potřeba použít operační systém reálného času. Proto je vhodné se nejprve zaměřit na analýzu úkolů, jež má program plnit.

Z hlediska generování pulsů je na časovou přesnost citlivá především GPIO varianta, v rámci níž se musí neustále kontrolovat hodnota hardwarového časovače, a při dosažení definovaných okamžiků je potřeba okamžitě vypínat GPIO piny. Uvedenou akci je navíc třeba opakovat v pravidelných intervalech. Řešení pomocí PWM rozhraní je v tomto ohledu méně náročné; kritická situace nastává pouze v případě změny pozice serv, kdy je potřeba postupně upravovat šířky ovládacích pulsů na požadované hodnoty. Obě varianty ovládání serv lze nicméně bez problémů implementovat formou prostého kódu. Situace se však začne komplikovat, jakmile dojde k začlenění meziprocesorové komunikace, která z podstaty věci probíhá asynchronně. Čas potřebný ke zpracování přijaté zprávy je navíc relativně dlouhý, a může tak snadno nastat situ-



Obrázek 15: Blokové schéma programu pro jádro Cortex M4.

ace, kdy namísto prioritní obsluhy serv stále probíhá analýza zprávy s novými šířkami pulsů. Dojde-li však k nasazení preemptivního RTOS, v rámci kterého je možné jednotlivým úlohám přiřadit vhodnou prioritu, lze veškerou starost s přepínáním úloh přenechat vestavěnému plánovači. Nejvyšší prioritu přitom bude mít úloha obsluhující výstupní signály pro serva, zatímco zpracování RPMsg zpráv a případné podpůrné úkoly mohou být zpracovány s nižší prioritou.

Schématické znázornění navrženého programu je k dispozici na obr. 15. Před samotným ovládáním serv musí dojít k inicializaci funkčních komponent. Toho je dosaženo provedením následujících kroků:

1. Nejprve je nutné inicializovat hardware, což mj. obnáší konfiguraci RDC modulu kvůli získání výhradního přístupu k používaným komponentám SoC. Dále jsou nastaveny výstupní piny do požadovaného GPIO nebo PWM módu.

2. Po dokončení inicializace hardwaru se na vyhrazené místo ve sdílené paměti (proměnná `TRACE_FLAGS`) zapíše informace o úspěšném startu programu. Tento krok slouží především k poskytnutí zpětné vazby nástroji `mqx_upload_on_m4SoloX`, který se používá pro

nahrání programu do paměti. Uvedený notifikační mechanismus nemá žádný zásadní vliv na funkčnost celého řešení a šlo by jej vynechat. Při jeho absenci je však třeba počítat s tím, že nástroj pro nahrávání kódu vypíše chybovou hlášku a skončí s nenulovou návratovou hodnotou.

3. Následně dochází k vytvoření úloh, jež poběží v rámci realtime systému FreeRTOS. Poté je spuštěn plánovač RTOS, který zajistí jejich korektní střídání.

V případě implementovaného ovladače serv je veškerá funkcionalita obstarávána pomocí trojice úloh: s nejvyšší prioritou 3 běží kód `GpioTask`, příp. `PwmTask` pro obsluhu výstupních pinů. Volba konkrétní varianty je dána v době kompilace a nelze ji již poté změnit. Jedná se o kritickou úlohu z hlediska správného, a především bezpečného chování celého systému, neboť pomocí robotické ruky dochází k interakci s okolním prostředím. Detailnějším popisem této úlohy se zabývají následující podkapitoly.

S nižší prioritou 2 běží relativně jednoduchá úloha `ExitTask` umožňující korektní ukončení programu. Úloha ve 100ms intervalech zkoumá obsah proměnné `BYTE_FOR_M4STOP` ve sdílené paměti, a při zjištěném požadavku na ukončení běhu kódu provede potřebné úkony k odstavení serv. Potvrzení o skončení běhu je následně signalizováno pomocí proměnné `TRACE_FLAGS`. V současnosti je tato úloha používána výhradně ve spolupráci s nástrojem `mqx_upload_on_m4SoloX`, který před nahráním nového kódu provádí notifikaci zápisem do proměnné `BYTE_FOR_M4STOP`. Jelikož se však v rámci produkčního nasazení počítá s perzistentním během kódu pro ovládání serv, nachází úloha `ExitTask` smysl pouze při vývoji a testování.

Nejnižší prioritu 1 má pak úloha `RpmsgTask`, která slouží k příjmu požadavků na nové pozice serv. Její grafické znázornění lze nalézt na obr. 16. Nejprve je provedena inicializace lokálního konce RPMsg kanálu. Poté probíhá v rámci nekonečné smyčky blokující čekání na příjem nové zprávy s následnou syntaktickou analýzou. V případě, že přišel požadavek na zaslání aktuální polohy serv, odesílá se tato informace aplikaci RoboticArm prostřednictvím RPMsg kanálu. Pokud však došlo k obdržení požadavku na nové pozice serv, dojde k jeho zařazení na konec fronty zpráv `xQueue`. Odtud si požadavky vyzvedává úloha `GpioTask` / `PwmTask` a postupně je zpracovává.

3.3.2 Programové řešení s GPIO

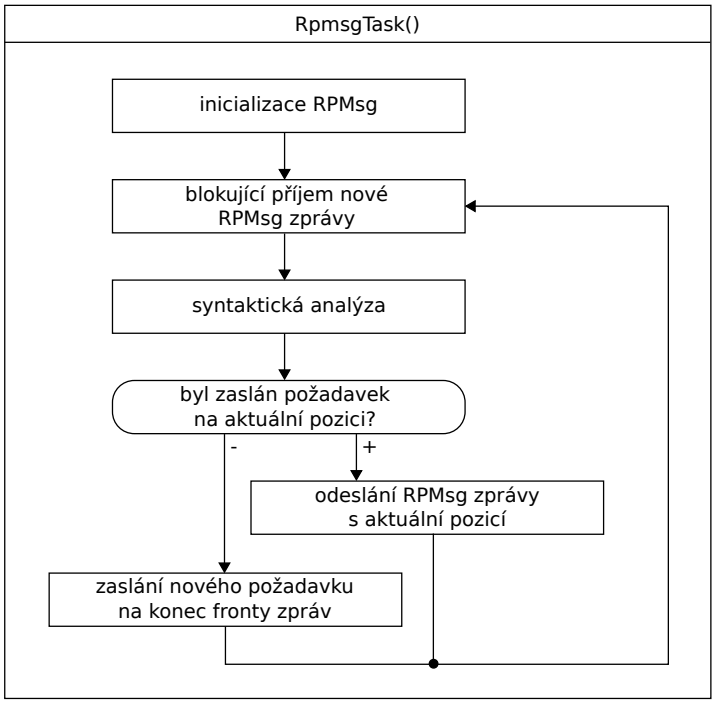
Při návrhu komponenty pro ovládání serv byl nejprve zvolen přístup pomocí ručního zapínání a vypínání GPIO pinů. Hlavním důvodem tohoto rozhodnutí byla především skutečnost, že pro práci s GPIO je díky dodávanému BSP balíku k dispozici API rozhraní.

Jednotlivé GPIO piny jsou v rámci API popsány pomocí struktury `gpio_config_t`. Ta obsahuje nezbytné informace pro nastavení pinu do GPIO módu (atributy `muxReg`, `muxConfig`) a zároveň i konfigurační parametry pro GPIO (atributy `padConfig`, `base`, `pin`). Hodnoty uvedených atributů lze pro použité piny nalézt v tab. 3. Pro detailní informace ohledně možností konfigurace pinů je čtenář odkázán na dokumentaci [7].

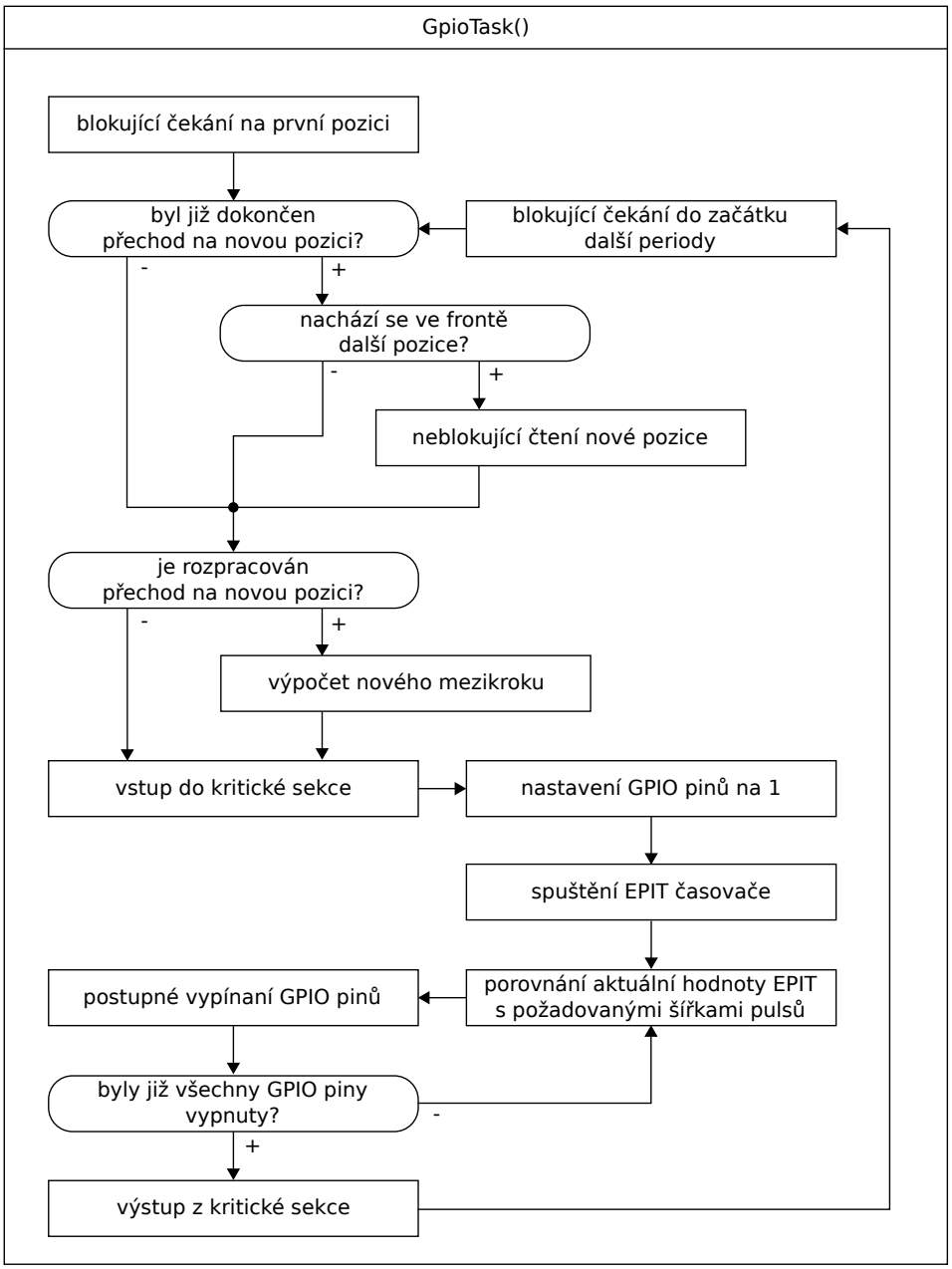
Tabulka 3: Konfigurace GPIO pinů.

Číslo pinu	Atributy datové struktury `gpio_config_t`					
	muxReg	muxConfig	padReg	padConfig	base	pin
3	RGMII2_RD2		RGMII2_RD2			14
4	RGMII2_RD3		RGMII2_RD3	SPEED(2) |		15
5	RGMII2_RD1	5	RGMII2_RD1	PKE_MASK |	GPIO5	13
6	RGMII2_RD0		RGMII2_RD0	DSE(6)		12
7	RGMII2_TD3		RGMII2_TD3			21

Pro odměření doby, po kterou mají být GPIO piny zapnuty, slouží jeden z dvojice dostupných časovačů EPIT. U něj byl zvolen zdroj hodinového signálu spolu s hodnotou přeškálování tak, aby bylo získáno časové rozlišení 1 µs, které je pro uvažované použití více než dostatečné. Jelikož EPIT umožňuje i pravidelné generování přerušení, zvažovala se původně možnost tímto způsobem oznamovat uplynutí mikrosekundového intervalu. Řídící logika pro vypínání GPIO by se v tomto případě nacházela v rámci obsluhy přerušení. Vzhledem k frekvenci jádra Cortex M4 a požadovanému časovému rozlišení by však zbylo na zpracování přerušení přibližně 200 instrukcí. Z tohoto důvodu byl zvolen poněkud jiný přístup: EPIT je nejprve nastaven na časový interval odpovídající maximální nastavené délce pulsu, a poté se časovač nechá asynchronně běžet, přičemž se ve smyčce porovnává uplynulý čas s požadovanými šířkami pulsů pro jednotlivé piny. Pokud dojde k překročení doby generování pulsu, GPIO pin je vypnut.



Obrázek 16: Schématické znázornění úlohy pro RPMsg komunikaci.



Obrázek 17: Schématické znázornění úlohy pro ovládání serv pomocí GPIO.

Představený postup má však jeden zásadní problém. Neustálá kontrola aktuální hodnoty časovače je narušována obsluhami přerušení, především těch, které jsou generovány plánovačem úloh samotného FreeRTOSu. Proto je nutné během odměřování pulsů dočasně pozastavit zpracování přerušení.

Na obr. 17 je čtenáři dán k dispozici diagram úlohy `GpioTask`. Přestože již předchozí text z velké části popsal její fungování, je vhodné se ještě zmínit o několika skutečnostech. Nové požadavky na polohy serv úloha získává čtením dat z fronty zpráv. Tato fronta je implemen-

tována v rámci systému FreeRTOS, přičemž data jsou do ní posílána úlohou `RpmsgTask`. Příjem první pozice serv se provádí pomocí blokujícího čekání, neboť v tu chvíli ještě není potřeba řešit zapínání a vypínání GPIO pinů. V případě obdržení nového požadavku pak dochází k postupnému přechodu na požadované pozice po 20ms krocích. Přijde-li např. informace o tom, že má být pohyb dokončen za 600 ms, je potřeba provést 30 iterací s lineární změnou šířek řídících pulsů. V rámci každé iterace přitom dochází k samotnému generování řídících pulsů. Nejprve se pomocí makra `taskENTER_CRITICAL` [18] dočasně zakáže obsluha přerušení. Následně dojde k nastavení výstupů GPIO pinů na hodnotu 1 a spuštění časovače EPIT, přičemž se pomocí busy waiting techniky kontroluje uplynulý čas a postupně se vypínají piny. Jakmile jsou všechny piny vypnuty, obnoví se pomocí makra `taskEXIT_CRITICAL` obsluha přerušení a vyčkává se na další časové okno. Po dokončení přechodu do cílové polohy se již pouze udržují požadované šířky pulsů a zkoumá stav fronty zpráv. Obsahuje-li fronta další požadavek, provede se neblokující čtení nové zprávy a následně se zahájí přechod na nové pozice serv.

U právě popsaného přístupu je vážným problémem přítomnost busy waitingu. V případě maximální šířky ovládacího pulsu 2,5 ms se ztrácí čekáním ve smyčce $\frac{2,5}{20} \cdot 100 \approx 13\,\%$ procesorového času. Jako vhodnější se proto ukazuje varianta, která pro generování pulsů používá PWM rozhraní. Řešení pomocí GPIO se však stále může hodit v případě absence PWM rozhraní, např. při výměně desky.

3.3.3 Řešení s využitím PWM rozhraní

Hlavní předností PWM přístupu je skutečnost, že není potřeba ručně přepínat výstupní piny mezi polohami zapnuto a vypnuto. V případě pulsní modulace stačí pouze nastavit dobu zapnutí a periodu pulsu; o samotné generování signálu se pak stará dedikovaný obvod. Použitý SoC přitom disponuje osmi PWM výstupy, které jsou v rámci desky UDOO Neo vyvedeny na vnitřní řadu pinů. Jediným problémem při použití tohoto přístupu tak zůstává skutečnost, že v rámci dodávaného BSP není implementováno API pro ovládání pulsní modulace. Proto bylo potřeba vytvořit alespoň základní množinu funkcí pro práci s registry PWM zařízení.

Tabulka 4: Konfigurace PWM pinů.

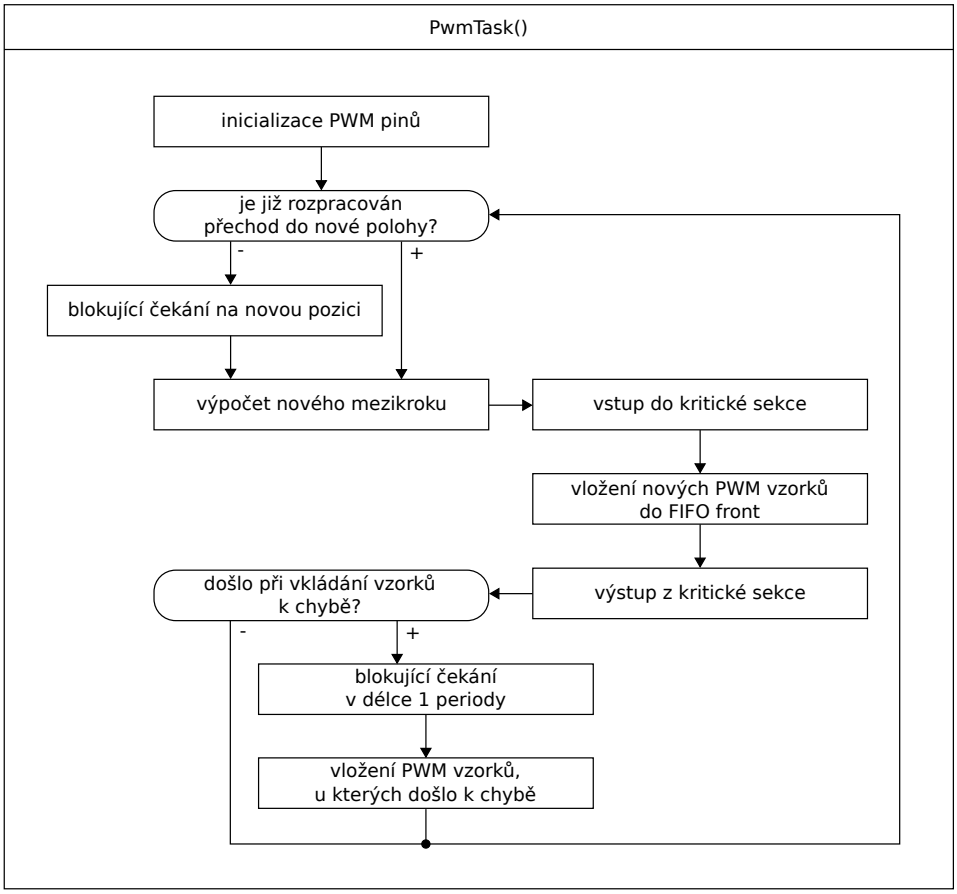
Číslo pinu	Atributy datové struktury `pwm_config_t`				
	muxReg	muxConfig	padReg	padConfig	pwmReg
3	RGMII2_RD2	2	RGMII2_RD2		PWM2
4	RGMII2_RD3	2	RGMII2_RD3	SPEED(2) |	PWM1
5	RGMII2_RD1	2	RGMII2_RD1	PKE_MASK |	PWM3
6	RGMII2_RD0	2	RGMII2_RD0	DSE(6)	PWM4
7	RGMII2_TD3	3	RGMII2_TD3		PWM5

V analogii s dříve popsanou strukturou `gpio_config_t` došlo k implementaci struktury `pwm_config_t`, která obsahuje atributy pro přenastavení pinu do PWM role spolu s adresou

asociovaného PWM registru. Obsah struktury je pro použité piny uveden v tab. 4. Vedle toho byly vytvořeny funkce pro usnadnění práce s PWM:

- `PWM_Ctrl_InitPin(pwm_config_t *pin)`: inicializace PWM registrů, volba zdroje hodinového signálu, konfigurace přeškálování,
- `PWM_SetSample(pwm_config_t *pin, uint16_t width)`: nastavení požadované šířky ovládacího pulsu [µs],
- `PWM_SetPeriod(pwm_config_t *pin, uint16_t period)`: nastavení požadované periody PWM signálu [µs],
- `PWM_EnablePin(pwm_config_t *pin)`: zapnutí generování PWM signálu,
- `PWM_DisablePin(pwm_config_t *pin)`: vypnutí generování PWM signálu.

Je třeba podotknout, že představené funkce nemají ambici stát se součástí generického API, neboť byly implementovány s důrazem na tuto práci. Při inicializaci pinů tak např. nelze zvolit libovolný zdroj hodinového signálu; vše je již nastaveno s ohledem na korektní ovládání serv.



Obrázek 18: Schématické znázornění úlohy pro ovládání serv pomocí PWM.

Samotná realizace úlohy `PwmTask` je v porovnání s GPIO variantou výrazně jednodušší, viz schéma na obr. 18. V podstatě se dá říci, že hlavní úkol kódu spočívá v příjmu nové polohy z fronty zpráv a výpočtu mezikroků kvůli přesunu konstantní rychlostí. Jednotlivé mezikroky jsou přitom pomocí funkce `PWM_SetSample` průběžně vkládány ke zpracování do FIFO fronty, jež je součástí PWM systému a umožňuje uchovat až čtyři vzorky. V případě zaplnění fronty pak stačí počkat jednu periodu, a vzorek následně vložit na uvolněné místo.

4 Test modifikovaného řešení

Následující kapitola se zaměřuje na verifikaci správného chování navrženého řešení. Celkem jsou provedeny čtyři testy, v rámci nichž se zkouší funkčnost RPMsg komunikace, simulace pádu GUI aplikace, korektní generování výstupních pulsů a celkové chování systému.

4.1 Meziprocesorová komunikace

V prvním testovacím scénáři se zkoumá funkčnost komunikace mezi oběma softwarovými komponentami. Kvůli tomu byl kód pro jádro Cortex M4 zkompilován se symbolem `DEBUG`, díky čemuž se na výstupu sériové linky zobrazují podrobné informace o právě prováděných činnostech. Program je pomocí nástroje `mqx_upload_on_m4SoloX` nahrán do paměti, načež dochází k vytvoření komunikačního kanálu pro RPMsg. O tom se lze snadno přesvědčit pomocí logu linuxového jádra, viz výpis 3.

Výpis 3: Výstup příkazu `dmesg`.

```
udooer@udooneo:~$ dmesg
...
[   20.919039] rpmsg_virtio RX: 00 00 00 00 35 00 00 00 00 00 00 00 28 00 00 00 ....5.......(...
[   20.919058] rpmsg_virtio RX: 72 70 6d 73 67 2d 6f 70 65 6e 61 6d 70 2d 64 65 rpmsg-openamp-de
[   20.919070] rpmsg_virtio RX: 6d 6f 2d 63 68 61 6e 6e 65 6c 00 00 00 00 00 00 mo-channel......
[   20.919078] rpmsg_virtio RX: 00 00 00 00 00 00 00 00                         ........
[   20.919093] NS announcement: 72 70 6d 73 67 2d 6f 70 65 6e 61 6d 70 2d 64 65 rpmsg-openamp-de
[   20.919104] NS announcement: 6d 6f 2d 63 68 61 6e 6e 65 6c 00 00 00 00 00 00 mo-channel......
[   20.919112] NS announcement: 00 00 00 00 00 00 00 00                         ........
[   20.919134] virtio_rpmsg_bus virtio0: creating channel rpmsg-openamp-demo-channel addr 0x0
[   20.919364] imx_rpmsg_tty rpmsg0: new channel: 0x400 -> 0x0!
[   20.919492] rpmsg_virtio TX: 00 04 00 00 00 00 00 00 00 00 00 00 0c 00 00 00 ................
[   20.919503] rpmsg_virtio TX: 68 65 6c 6c 6f 20 77 6f 72 6c 64 21             hello world!
[   20.922122] Install rpmsg tty driver!
```

V logu je zaznamenáno úspěšné vytvoření meziprocesorového komunikačního kanálu, na jehož jednom konci je program pro jádro Cortex M4, zatímco na opačném sídlí jaderný modul `imx_rpmsg_tty`. Následně je provedena simulace zaslání nové pozice serv, a to prostřednictvím přímého zápisu do znakového zařízení `/dev/ttyRPMSG`:

```
udooer@udooneo:~$ echo '#0 P1400 #1 P1800 #4 P2000 T600\r' > /dev/ttyRPMSG .
```

Jak ukazuje výpis 4, na výstupu sériové linky jádra Cortex M4 se objeví potvrzení o přijetí nové pozice spolu s výpisem informací o prováděném pohybu. Tímto byla otestována průchodnost RPMsg kanálu pro meziprocesorovou komunikaci.

Výpis 4: Ladící výstup programu pro ovládání serv.

```
DEBUG: xQueue allocation
DEBUG: PWM init
DEBUG: Calling FreeRTOS scheduler
RPMSG Init as Remote
Name service handshake is done, M4 has setup a rpmsg channel [0 ---> 1024]
```

```
Got Message From Master Side at addr (1024): "#0 P1400 #1 P1800 #4 P2000 T600\r\n" [len : 35]
DEBUG(RPMsg): Parsing new position: #0 P1400 #1 P1800 #4 P2000 T600\r\n
DEBUG(parse_position): Got new servo: 0
DEBUG(parse_position): Got new pulse: 1400
DEBUG(parse_position): Got new servo: 1
DEBUG(parse_position): Got new pulse: 1800
DEBUG(parse_position): Got new servo: 4
DEBUG(parse_position): Got new pulse: 2000
DEBUG(parse_position): Got new TTF: 600
DEBUG(RPMsg): Enqueuing new position [1400, 1800, 0, 0, 2000, 0]
DEBUG(PWM): Starting transition to a new position: [1400, 1800, 0, 0, 2000, 0]
T = 600
...
Iter: 30/30
DEBUG(PWM): New intermediary position: [1400, 1800, 0, 0, 2000, 0]
DEBUG(PWM): Final position reached
DEBUG(PWM): Entering critical section
DEBUG(PWM): Leaving critical section
```

4.2 Restart aplikace RoboticArm

Následující scénář zkoumá chování systému při ukončení a opětovném spuštění aplikace RoboticArm. Poskytnutá verze aplikace se totiž dokáže při startu zeptat protistrany na aktuální pozice serv, a to zasláním zprávy obsahující konfiguraci nevyužitého kanálu č. 7. Pokud ještě nebyla serva inicializována, vrací se aplikaci řetězec `"noInit"`. V takovém případě dochází k přenastavení robotické ruky do výchozí polohy. Při spouštění aplikace RoboticArm však již serva mohou mít nastaveny pozice. K uvedené situaci může dojít, pokud GUI aplikace běžela v minulosti a byla následně ukončena. V takovém případě se aplikaci vrací zpráva o aktuální pozici serv. Formát odpovědi je přitom shodný s tvarem, v němž jsou zasílány požadavky na nové pozice serv.

Cílem prvního testu je vyzkoušení chování aplikace pro případ neinicializovaných serv. Z tohoto důvodu byl program pro jádro Cortex M4 opětovně nahrán do paměti a následně došlo ke spuštění grafické aplikace RoboticArm, viz výpis 5.

Výpis 5: Výstup aplikace RoboticArm při neinicializovaných servech.

```
udooer@udooneo:~/RoboticArm$ ./RoboticArm
 write succeess!
waiting for input
Response: noInit
 7
Read succeess! noInit
 : 7
Response:noInit
; fresh 0
checking response
no values
```

Ze standardního výstupu aplikace lze usoudit, že program úspěšně přijal zprávu o neinicializovaných servech. Nyní je na řadě otestování zpětného zasílání informace o polohách iniciali-

zovaných serv. Proto nejprve došlo k přenastavení robotické ruky do libovolně zvolené polohy a poté byla aplikace RoboticArm ukončena. Jak je vidět ve výpisu 6, po opětovném spuštění programu dochází k předání informace o aktuální konfiguraci serv, z čehož si aplikace dokáže dopočítat pozici robotické ruky v prostoru.

Výpis 6: Výstup aplikace RoboticArm v případě inicializovaných serv.

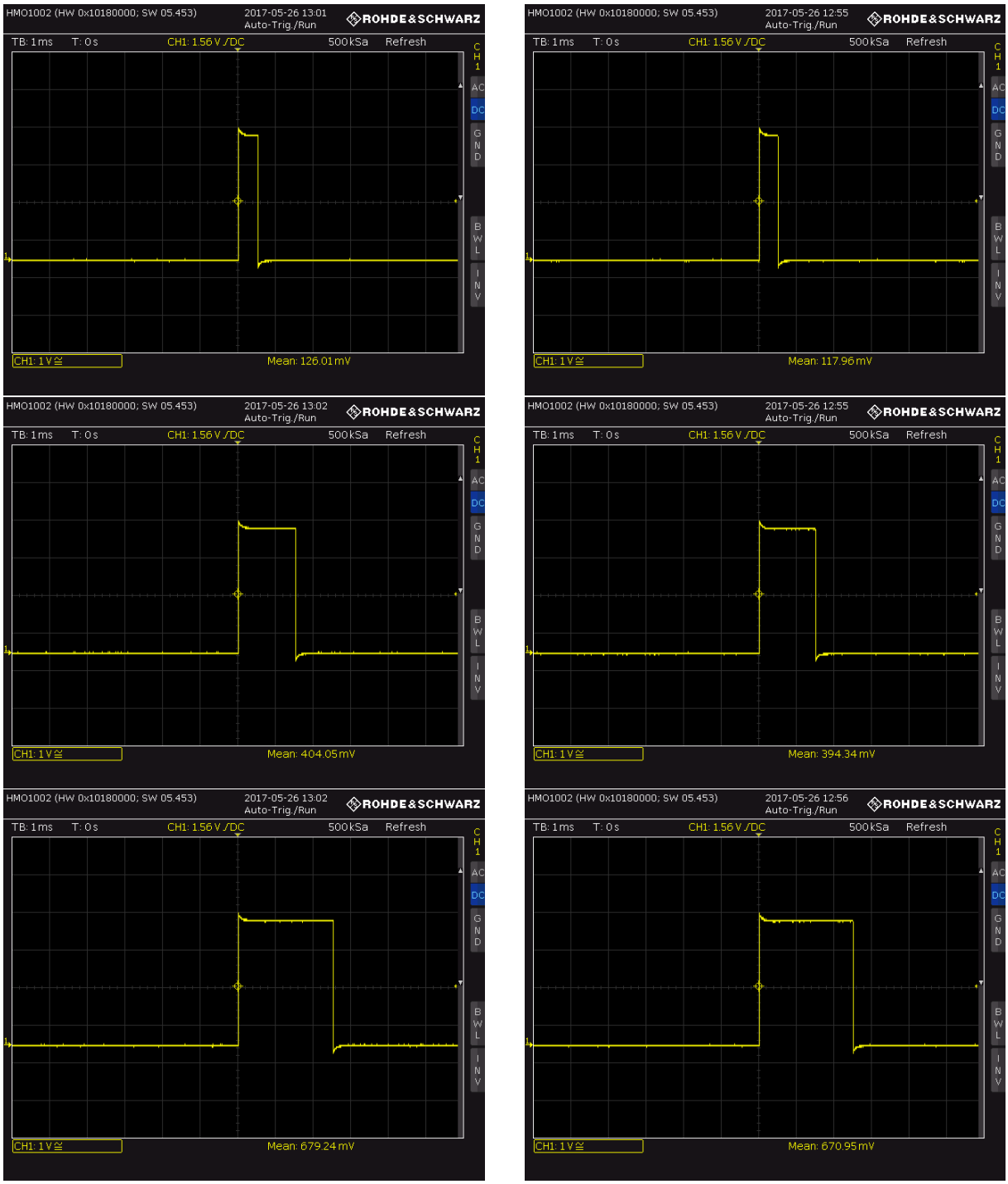
```
udooer@udooneo:~/RoboticArm$ ./RoboticArm
 write succeess!
waiting for input
Response: 0 P1400 1 P1493 2 P1413 3 P1110 4 P1235 5 P0 T3000
 51
Read succeess! 0 P1400 1 P1493 2 P1413 3 P1110 4 P1235 5 P0 T3000
 : 51
Response:0 P1400 1 P1493 2 P1413 3 P1110 4 P1235 5 P0 T3000
; fresh 0
checking response
values
Set values readed from M4
servo0 val:1400
servo1 val:1493
servo2 val:1413
servo3 val:1110
servo4 val:1235
u0:90 u1:-4 u2:49 u3:-35 u4:10
FINAL ASSIGMENTSX:222.032 Y:90 Z:146.803 A:2 A2:10
Solved z146.803 x222.032 a2 a2:10
Solved z146 x222 a2 a2:10
 #1 P1493 #17 P1493 #2 P1403 #3 P1110 T3000
move 0
 #1 P1493 #17 P1493 #2 P1403 #3 P1110 T3000
```

Na základě provedených testů lze konstatovat, že mezi oběma softwarovými komponentami dochází ke korektní výměně informací.

4.3 Generování výstupních pulsů

V rámci dalšího scénáře došlo k otestování průběhu výstupních signálů. K tomuto účelu byly generovány pulsy o různých šířkách, přičemž jejich kvalita byla následně ověřena pomocí osciloskopu.

Zachycené tvary pulsů jsou pro oba implementované mechanismy k dispozici na obr. 19. Na základě výstupu z osciloskopu lze usoudit, že průběhy řídících signálů serv odpovídají nastaveným hodnotám šířky pulsu. Z hlediska časové stability pak vychází o něco lépe PWM varianta, neboť u GPIO řešení byla zjištěna nepatrná fluktuace šířky pulsu. Uvedená nedokonalost není v případě řešeného problému na škodu, u citlivějších systémů by ale mohla znemožňovat použití GPIO implementace.

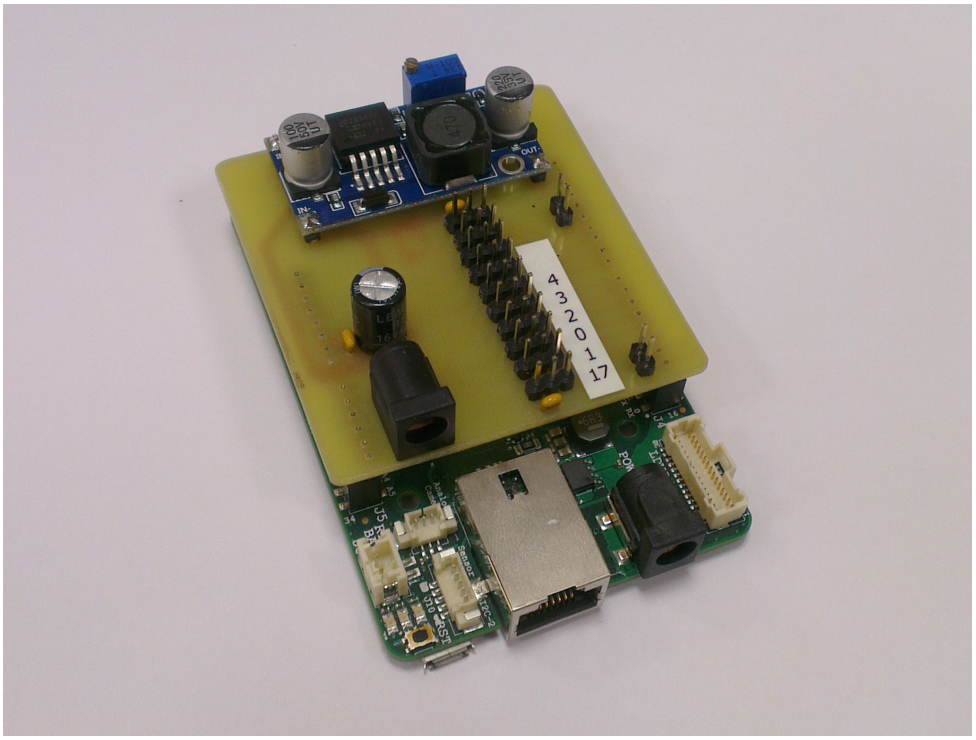


(a) GPIO **(b)** PWM

Obrázek 19: Průběh řídících pulsů pro a) GPIO a b) PWM variantu. V experimentu byly postupně nastaveny šířky pulsu 500, 1500 a 2500 µs (shora).

4.4 Ovládání robotické ruky

Poslední test slouží ke zhodnocení celkového chování systému pro ovládání robotické ruky. Původní deska i.MX Sabre SD byla proto spolu s modulem SSC-32 nahrazena řešením UDOO Neo, na kterém běžel modifikovaný systém Udoobuntu a ovládací program pro jádro Cortex M4 v PWM režimu. Pro snažší připojení serv byla vedoucím práce vyvinuta rozšiřující deska, která zajišťuje i napájení celého systému. Rozšiřující deska je vyobrazena na obr. 20 a její schéma zapojení spolu s DPS čtenář nalezne v příloze C.



Obrázek 20: UDOO Neo s rozšiřující deskou pro připojení serv.

Na přiloženém elektronickém médiu je k dispozici videozáznam s ukázkou ovládání ruky pomocí modifikovaného řešení (soubor `roboticarm.mp4`). Lze konstatovat, že systém funguje korektně. Vyskytly se pouze dva dílčí nedostatky:

- Při hře „Člověče, nezlob se!“ docházelo k nepřesnému umisťování figurky po hrací ploše; figurka obvykle skončila půl centimetru mimo požadované místo. Původně padlo podezření na špatnou kalibraci serv. Nakonec se však ukázalo, že tato nesrovnalost byla způsobena nepřesnými souřadnicemi políček na hrací ploše, které jsou v rámci aplikace RoboticArm specifikovány v externím souboru `GamePositions`.
- V módu *Position* se při ovládání pozice robotické ruky projevuje jistá necitlivost grafického rozhraní; pro nastavení nové pozice je obvykle nutné vícenásobné kliknutí. Autorovi se nepovedlo zjistit příčinu problému, podezření však padá na odlišnou verzi knihovny Qt.

5 Závěr

Práce se zabývala možnostmi vylepšení systému pro ovládání robotické ruky. Z tohoto důvodu byl čtenář v úvodních kapitolách nejprve seznámen s výchozími hardwarovými komponentami. Poté následoval popis aplikace RoboticArm, díky které lze ovládat polohu robotické ruky prostřednictvím grafického rozhraní.

Navržené úpravy byly představeny v rámci třetí kapitoly. Především se jednalo o výměnu řídící desky, kde původní platformu i.MX Sabre SD nahradila běžně dostupná deska UDOO Neo. Důvodem uvedeného kroku bylo především získání možnosti přímého ovládání serv bez nutnosti použití modulu SSC-32. Pro snadné připojení serv byla navíc vedoucím práce zhotovena rozšiřující deska, která se nasazuje přímo na piny UDOO Neo. Uvedené řešení má zároveň i pozitivní vedlejší efekt: díky kompaktním rozměrům desky a absenci SSC-32 dosahuje nový řídící systém velikosti platební karty. Jelikož je hlavní komponentou desky UDOO Neo čip i.MX6 SoloX, byla rovněž stručně popsána architektura tohoto SoC s akcentem na obě výpočetní jednotky Cortex A9 a Cortex M4.

Z hlediska softwarové části došlo k rozložení celkové funkcionality do dvou celků: na jádru Cortex A9 běží linuxová distribuce s aplikací RoboticArm, která pomocí mechanismu meziprocesorové komunikace předává požadavky na změnu polohy robotické ruky realtime kódu, jenž běží na jádru Cortex M4 a generuje řídící signály pro ovládání serv. V rámci linuxové části bylo rozhodnuto ponechat distribuci Udoobuntu, která byla vytvořena přímo pro desku UDOO Neo. Jelikož však u distribučního jádra původně chyběla podpora mechanismu RPMsg pro meziprocesorovou komunikaci, muselo dojít k úpravě zdrojových kódů a následné rekompilaci jádra. V případě programu pro jádro Cortex M4 byl použit realtime operační systém FreeRTOS. Právě popsaný systém úspěšně absolvoval sérii testů, které se zaměřovaly jednak na chování jednotlivých komponent, tak i systému jako celku.

Přestože je navržený systém funkční, nabízí se několik možností, jak by jej šlo vylepšit. Komunikace mezi oběma softwarovými částmi v současnosti probíhá formou posílání textových řetězců, které jsou ve stejném tvaru jako zprávy pro kontrolér SSC-32. To znamená, že na odesílající straně se požadované polohy serv musí nejprve zakódovat do textové formy, přičemž přijímající strana musí získaný řetězec následně analyzovat. Jako elegantnější řešení se jeví použití mechanismu vzdáleného volání procedur eRPC [19], pomocí něhož bychom mohli přímo z aplikace RoboticArm volat funkci, která by se prováděla na straně jádra Cortex M4. Nasazení uvedeného komunikačního mechanismu by však znamenalo značný zásah do obou softwarových komponent, z časových důvodů bylo proto od něj nakonec upuštěno.

Rovněž by bylo vhodné mít možnost dočasného odpojení řídících signálů pro serva. Aktuální verze rozšiřující desky totiž napájí současně serva i samotné UDOO Neo, přičemž neexistuje možnost nouzového zastavení robotické ruky, aniž by nedošlo k vypnutí celého systému. V rámci náhradního řešení by mohla aplikace RoboticArm obsahovat tlačítko pro okamžité zastavení pohybu. Jelikož má tato informace vysokou prioritu, musí v rámci meziprocesorové komunikace

„přeskočit“ dosud nezpracované požadavky na nové pozice serv. V praxi by to znamenalo implementovat prioritizaci na úrovni RPMsg, což je však nad rámec této diplomové práce.

Literatura

[1] *Lynxmotion L6AC-KT Robotic Arm* [online]. c2008 [cit. 2017-04-03].
URL `http://www.lynxmotion.com/driver.aspx?Topic=assem01#l6`

[2] *Lynxmotion SSC-32 Servo Controller Board Manual* [online]. V2.0, c2005 [cit. 2017-04-01].
URL `http://www.lynxmotion.com/images/data/ssc-32.pdf`

[3] *i.MX Linux® User's Guide*. Freescale Semiconductor, Document Number: IMXLUG, Rev. L3.14.28_1.0.0-ga, Apr. 2015.

[4] *UDOO Neo Documentation* [online]. c2015, poslední revize 11. 3. 2017 [cit. 2017-04-01].
URL `http://www.udoo.org/docs-neo/Introduction/Introduction.html`

[5] *UDOO Neo Schematics* [online]. c2015 [cit. 2017-04-09].
URL `http://www.udoo.org/download/files/schematics/UDOO_NEO_schematics.pdf`

[6] *i.MX 6SoloX Power Consumption Measurement*. Freescale Semiconductor, Document Number: AN5050, Rev. 0, May 2015.

[7] *i.MX 6SoloX Applications Processor Reference Manual*. NXP Semiconductors, Document Number: IMX6SXRM, Rev. 1, 6/2016.

[8] HALLINAN, Christopher. *Embedded Linux primer: a practical real-world approach, 2nd ed.* Upper Saddle River, NJ: Prentice Hall, c2011. ISBN 978-0-13-701783-6.

[9] *Freescale MQX™ RTOS User's Guide*. Freescale Semiconductor, Document Number: MQXUG, Rev. 14, Apr. 2015.

[10] BARRY, Richard. *Using the FreeRTOS Real Time Kernel - a Practical Guide - Cortex M3 Edition*. Real Time Engineers Ltd, c2010. ISBN 978-1-4461-7030-4.

[11] *FreeRTOS BSP i.MX 6SoloX API Reference Manual*. Freescale Semiconductor, Document Number: FRTOS6XAPIRM, Rev. 0, Dec 2015.

[12] *Das U-Boot – the Universal Boot Loader* [online]. Verze 1.29, poslední revize 20. 4. 2017 [cit. 2017-04-23].
URL `https://www.denx.de/wiki/U-Boot`

[13] *MultiCore Communication Library – User Guide*. Freescale Semiconductor, Document Number: MQXMCCUG, Rev. 1.2, Feb. 2014.

[14] *OpenAMP – Open-source software framework for developing AMP systems application software* [online]. c2016, poslední revize 16. 12. 2016 [cit. 2017-04-23].
URL `https://github.com/OpenAMP/open-amp/wiki`

[15] RUSSELL, Rusty. *Virtio: Towards a De-Facto Standard For Virtual I/O Devices*. ACM SIGOPS Operating Syst. Review 42:95–103, c2008.

[16] *RPMSG RTOS Layer User's Guide*. Freescale Semiconductor, Document Number: RPMSGRTOSLAYERUG, Rev. 0.1, Nov 2015.

[17] ŠRÁMEK, Jan. *Řízení robotické ruky pomocí Freescale i.MX6SX*. Bakalářská práce, katedra informatiky, FEI, VŠB-TUO, 2017.

[18] BARRY, Richard. *Mastering the FreeRTOS Real Time Kernel - a Hands-On Tutorial Guide*. Real Time Engineers Ltd, c2015. Pre-release 15114, ISBN nebylo v době psaní práce dosud přiřazeno.

[19] *eRPC API Reference 1.3.0* [online]. c2016 Freescale Semiconductor [cit. 2017-04-28]. URL `https://embeddedrpc.github.io/index.html`.

A Obsah přiloženého média

/	
├─ text/	
│ └─ dp.pdf	elektronická verze textu DP
├─ media/	
│ └─ roboticarm.mp4	videoukázka ovládání robotické ruky
├─ source/	
│ ├─ kernel/	
│ │ ├─ linux_kernel.zip	zdrojové kódy distribučního jádra
│ │ └─ add_rpmsg_support.patch	patch pro zprovoznění RPMsg
│ ├─ m4/	
│ │ ├─ FreeRTOS_BSP_1.0.0_iMX6SX/	FreeRTOS port pro SoC i.MX6 SoloX
│ │ └─ robotic_arm/	zdrojové kódy programu pro jádro Cortex M4
│ └─ RoboticArm/	zdrojové kódy Qt aplikace RoboticArm
└─ build/	
├─ robotic_arm_gpio.bin	předkompilovaná GPIO varianta programu pro Cortex M4
└─ robotic_arm_pwm.bin	předkompilovaná PWM varianta programu pro Cortex M4

B Postup při modifikaci výchozí instalace Udoobuntu

Následující příloha shrnuje nezbytné kroky, které je třeba vykonat pro zajištění funkčnosti všech komponent potřebných pro ovládání robotické ruky. Jedná se především o zprovoznění RPMsg subsytému v linuxovém jádru, úspěšné nahrání programu pro výpočetní jádro Cortex M4 a spuštění aplikace RoboticArm. Vycházíme přitom z čisté instalace systému Udoobuntu verze 2.1.2.

K provedení vybraných úprav je potřeba externích souborů. Ty jsou buď volně ke stažení, příp. je lze najít na médiu, jež je součástí této práce. Pokud se daný soubor nachází na přiloženém médiu, je ve specifikaci cesty použit speciální symbol `///`.

Některé úkony je třeba provádět přímo na běžícím systému UDOO Neo. Pak je tato skutečnost odlišena promptem `UDOO$` nebo `UDOO#` v případě nutnosti rootovského oprávnění. Provádí-li se daný příkaz na dedikovaném (a zpravidla mnohem výkonějším) systému, je použit prompt `PC$`.

B.1 Zprovoznění RPMsg

Jak již bylo zmíněno v textu diplomové práce, „oficiální“ verze linuxového jádra pro Udoobuntu neobsahuje RPMsg funkcionalitu. Budeme proto muset stáhnout zdrojové soubory výchozího jádra, aplikovat na ně patch a následně jádro zkompilovat. Na tomto místě je třeba zdůraznit, že samotnou kompilaci nebudeme spouštět na desce UDOO Neo, nýbrž na externím systému. Ten však s největší pravděpodobností nepoběží na architektuře ARM, a ve skutečnosti tak budeme provádět kroskompilaci. Nejde o nijak zásadní problém, bude ale potřeba mít k dispozici dodatečné nástroje, které nám umožní pracovat s odlišnou architekturou.

1. Nejprve je třeba nainstalovat nástroje pro kroskompilaci. V případě systémů založených na RHEL7 použijeme

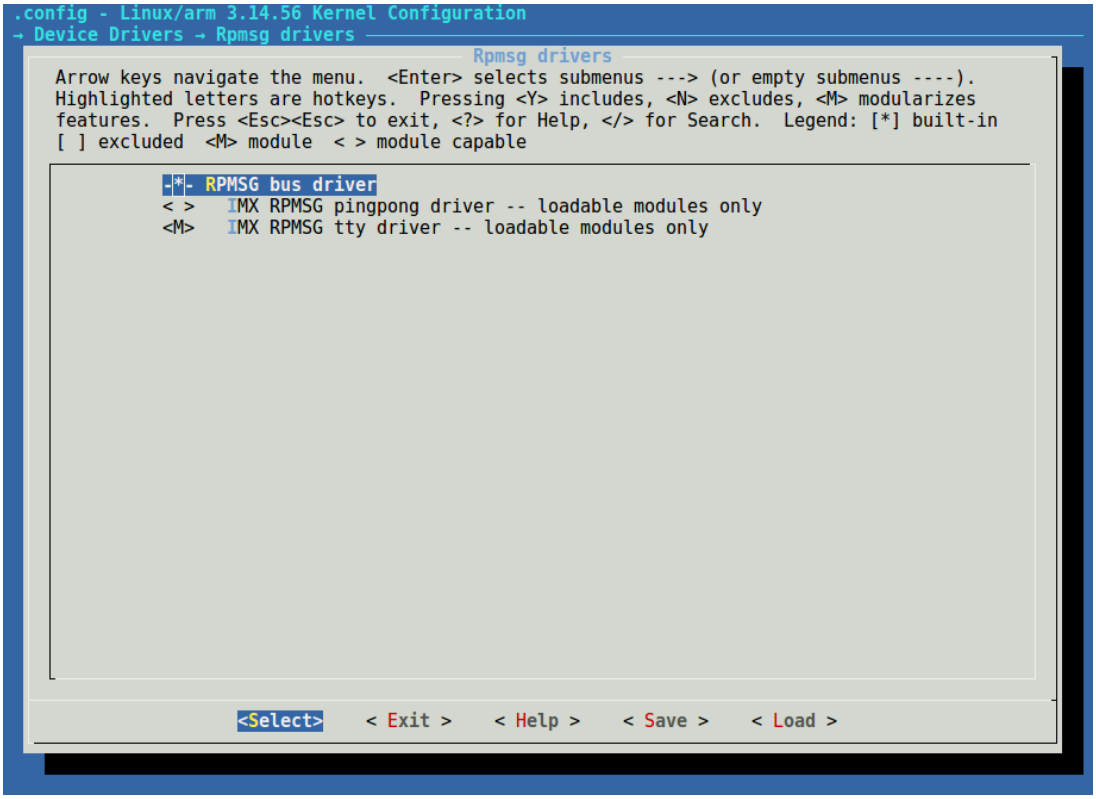
   ```
   PC$ sudo yum install gawk wget git diffstat unzip texinfo  \
         make automake gcc gcc-c++ kernel-devel chrpath socat \
         xterm picocom lzop gcc-arm-linux-gnu cross-gcc-common
   ```

   Dáváme-li však z jakéhokoliv důvodu přednost distribucím odvozeným od Debianu (např. Ubuntu, Mint), provedeme příkaz

   ```
   PC$ sudo apt-get install gawk wget git diffstat unzip texinfo \
        gcc-multilib build-essential chrpath socat libsdl1.2-dev \
        xterm picocom ncurses-dev lzop gcc-arm-linux-gnueabihf
   ```

2. Stáhneme si oficiální Git repozitář linuxového jádra pro UDOO Neo:

   ```
   PC$ git clone https://github.com/UDOOboard/linux_kernel
   ```



Obrázek 21: Povolení kompilace `tty` ovladače pro RPMsg v nástroji menuconfig.

3. Následně aplikujeme připravený patch pro získání RPMsg funkcionality. Je vhodné podotknout, že uvedený patch byl vytvořen pro verzi jádra 3.14.56 a autor není schopen garantovat jeho funkčnost i pro jiné verze. Navíc je možné, že budoucí verze jádra pro UDOO Neo již budou mít fungující RPMsg systém v základu, a tudíž nebude třeba provádět žádné dodatečné úpravy.

   Zkopírujeme patch do souborové struktury stáhnutého repozitáře:

   ```
   PC$ cp ///source/kernel/add_rpmsg_support.patch linux_kernel
   ```

   Následně přejdeme do adresáře `linux_kernel` a (volitelně) vytvoříme vlastní Git větev:

   ```
   PC$ cd linux_kernel
   PC$ git checkout -b add_rpmsg
   ```

   Před samotným použitím patche je vhodné nejprve otestovat, zdali nedojde k případným chybám:

   ```
   PC$ git apply --check add_rpmsg_support.patch
   ```

   Provede-li se předchozí příkaz bez jakéhokoli výpisu na terminál, můžeme bez obav patch aplikovat:

```
PC$ git apply add_rpmsg_support.patch
```

Aplikované změny je vhodné uložit prostřednictvím commitu, jinak bude mít zkompilované jádro ve svém názvu příponu „-dirty“:

```
PC$ git add -A
PC$ git commit
```

4. Od tohoto místa návod více méně kopíruje postup v oficiální dokumentaci [4]. Jako další krok tudíž vygenerujeme konfigurační soubor pro kompilaci jádra:

   ```
   PC$ ARCH=arm make udoo_neo_defconfig
   PC$ ARCH=arm make menuconfig
   ```

   Pomocí konfiguračního nástroje `menuconfig` přitom povolíme kompilaci jaderného modulu `imx_rpmsg_tty`. Položka se nachází v sekci *Device Drivers -> Rpmsg drivers -> IMX RPMSG tty driver*, viz též obr. 21.

5. Nyní se již můžeme pustit do kompilace jádra, device tree a jaderných modulů:

   ```
   PC$ ARCH=arm CROSS_COMPILE=arm-linux-gnu- make zImage -j4
   PC$ ARCH=arm CROSS_COMPILE=arm-linux-gnu- make dtbs -j4
   PC$ ARCH=arm CROSS_COMPILE=arm-linux-gnu- make modules -j4
   ```

   Předchozí série příkazů platí pro distribuce odvozené od RHEL7, v případě použití systému vycházejícího z Debianu nastavíme `CROSS_COMPILE=arm-linux-gnueabihf-`. Doba kompilace silně závisí na použitém hardwaru, ale obvykle se pohybuje v řádu jednotek minut. Její délku můžeme ovlivnit vhodnou volbou parametru `-j`, který udává počet příkazů, jež se budou pouštět současně (doporučuje se dvojnásobek počtu jader CPU).

6. Zkompilované jádro je nyní potřeba přenést na paměťovou kartu s nahranou distribucí Udoobuntu. V oficiálním návodu [4] se doporučuje připojit kartu do čtečky a zkopírovat jednotlivé soubory do cílových adresářů:

   ```
   PC$ BOOT_PARTITION=/dev/mmcblk0p1
   PC$ ROOT_PARTITION=/dev/mmcblk0p2
   PC$ cp arch/arm/boot/zImage $BOOT_PARTITION
   PC$ cp arch/arm/boot/dts/*.dtb $BOOT_PARTITION/dts
   PC$ ARCH=arm CROSS_COMPILE=arm-linux-gnu- make firmware_install \
               modules_install INSTALL_MOD_PATH=$ROOT_PARTITION
   ```

   Autorovi však uvedený postup nefungoval, neboť jeho operační systém (Fedora 24) měl problém se čtením superbloků na obou oddílech paměťové karty. Alternativní možností je zkopírovaní souborů na běžící systém pomocí `scp`:

```
PC$ ARCH=arm CROSS_COMPILE=arm-linux-gnu- make firmware_install \
            modules_install INSTALL_MOD_PATH=/tmp
PC$ scp arch/arm/boot/zImage root@udoo:/boot
PC$ scp -r arch/arm/boot/dts/*.dtb root@udoo:/boot/dts
PC$ scp -r /tmp/lib root@udoo:/lib
```

Předchozího postupu lze z pochopitelných důvodů využít pouze tehdy, má-li UDOO síťovou konektivitu a je-li povolen vzdálený přístup pro roota v konfiguraci ssh démona.

7. Po úspěšném zkopírování všech souborů na paměťovou kartu můžeme oba připojené oddíly odmountovat a kartu následně vložit do UDOO. Pokud jsme zvolili `scp` variantu, postačí rebootovat Udoobuntu.

8. O tom, zdali v rámci Udoobuntu skutečně používáme upravené jádro, se přesvědčíme pomocí nástroje `uname`:

   ```
   UDOO$ uname -r
   3.14.56-02055-ge1b41f8
   ```

9. V tuto chvíli je potřeba zavést modul `imx_rpmsg_tty` a rovněž zajistit, aby byl k dispozici i po rebootu systému:

   ```
   UDOO# modprobe imx_rpmsg_tty
   UDOO# echo imx_rpmsg_tty >> /etc/modules
   ```

10. Výše uvedený modul má na starosti obsluhu znakového zařízení `/dev/ttyRPMSG`, které se automaticky vytvoří po úspěšné inicializaci RPMsg kanálu mezi oběma výpočetními jádry. Ve výchozím stavu může toto zařízení používat pouze root. Jelikož však aplikace RoboticArm poběží pod neprivilegovaným uživatelem, musíme upravit pravidla pro udev. Nejprve vytvoříme nový soubor `/etc/udev/rules.d/90-tty-rpmsg.rules` s obsahem

    ```
    SUBSYSTEM=="tty", KERNEL=="ttyRPMSG", GROUP="udooer", MODE="0660"
    ```

    Následně necháme udev přečíst novou konfiguraci:

    ```
    UDOO# udevadm control --reload-rules
    ```

    V případě, že již došlo k vytvoření zařízení `/dev/ttyRPMSG`, bude potřeba odebrat a znova přidat jaderný modul:

    ```
    UDOO# modprobe -r imx_rpmsg_tty && modprobe imx_rpmsg_tty
    ```

11. Abychom zabránili nechtěnému přepisu modifikovaného jádra při pozdějším upgradu systému, nastavíme u vybraných balíčků příznak hold:

    ```
    UDOO# apt-mark hold linux-image-3.14.56-udooneo linux-3.14.56-udooneo \
                        linux-firmware-image-3.14.56-udooneo
    ```

B.2 Kompilace a nahrání programu pro jádro Cortex M4

Při vývoji softwaru pro architekturu ARM se obvykle používá plnohodnotné vývojové prostředí, jakým je např. DS-5 Development Studio[6]. Vzhledem k velikosti projektu pro ovládání robotické ruky nám však bude bohatě stačit samotný toolchain pro danou platformu.

1. Jelikož buildovací systém závisí na nástroji `cmake`, je potřeba mít nainstalovaný stejnojmenný balíček.

2. Z webových stránek `https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads` si stáhneme aktuální verzi toolchainu. Získaný archiv rozbalíme, a poté do proměnné prostředí `ARMGCC_DIR` nastavíme cestu k adresáři s toolchainem, např.:

   ```
   PC$ export ARMGCC_DIR=/opt/gcc-arm-none-eabi-5_4-2016q2/
   ```

3. Z přiloženého média si zkopírujeme zdrojové soubory programu pro ovládání robotické ruky:

   ```
   PC$ cp -R ///source/robotic_arm .
   ```

4. (Volitelně) provedeme úpravu zdrojového kódu. Především se jedná o to, zdali požadujeme PWM variantu programu – v tomto případě musí být definován symbol `USE_PWM`.

5. Přejdeme do adresáře `robotic_arm/armgcc` a spustíme kompilační skript:

   ```
   PC$ cd robotic_arm/armgcc
   PC$ ./build_release.sh
   ```

   V případě úspěchu by měl výstup na terminálu končit řádky

   ```
   ...
   Linking C executable release/robotic_arm.elf
   [100%] Built target robotic_arm
   ```

6. Získaný binární soubor, jenž se nachází v podadresáři `release/`, zkopírujeme na UDOO Neo:

   ```
   PC$ scp release/robotic_arm.bin udooer@udoo:/home/udooer
   ```

7. Nyní potřebujeme nahrát binární soubor do paměti. K tomu můžeme použít buď program `mqx_upload_on_m4SoloX`:

   ```
   UDOO$ mqx_upload_on_m4SoloX robotic_arm.bin


    UDOO Neo - MQX uploader v. 1.2.0
   ```

[6] `https://developer.arm.com/products/software-development-tools/ds-5-development-studio`

```
UDOO Neo - FILENAME = robotic_arm.bin; loadaddr = 0x84000000
UDOO Neo - start - end (0x84000000 - 0x84080000)
UDOO Neo - Waiting M4 Run, m4TraceFlags: 000001E0
UDOO Neo - M4 sketch is running!
```

nebo skript `udooneo-m4uploader`, který interně používá předchozí nástroj, a navíc zajišťuje automatické nahrávání binárního souboru při restartu systému:

```
UDOO$ udooneo-m4uploader robotic_arm.bin

Success!!
```

B.3 Zprovoznění GUI aplikace RoboticArm

Grafická aplikace RoboticArm je založena na Qt frameworku a k jejímu vytvoření bylo užito prostředí Qt Creator. Při kompilaci lze tudíž postupovat tak, že na externí systém nainstalujeme Qt Creator spolu s podpůrnými balíčky a následně provedeme kroskompilaci na architekturu ARM. Ukazuje se však, že zdaleka nejjednodušší cestou bude provést nativní kompilaci přímo v prostředí Udoobuntu.

1. Nejprve zkopírujeme zdrojové kódy aplikace na UDOO Neo:

   ```
   PC$ scp -r ///source/RoboticArm udooer@udoo:~
   ```

2. Následně doinstalujeme veškeré závislosti:

   ```
   UDOO# apt-get install qt5-default libgstreamer0.10-0 libgstreamer0.10-dev \
           libopencv-dev
   ```

3. Nyní již stačí přejít do adresáře se zdrojovými soubory:

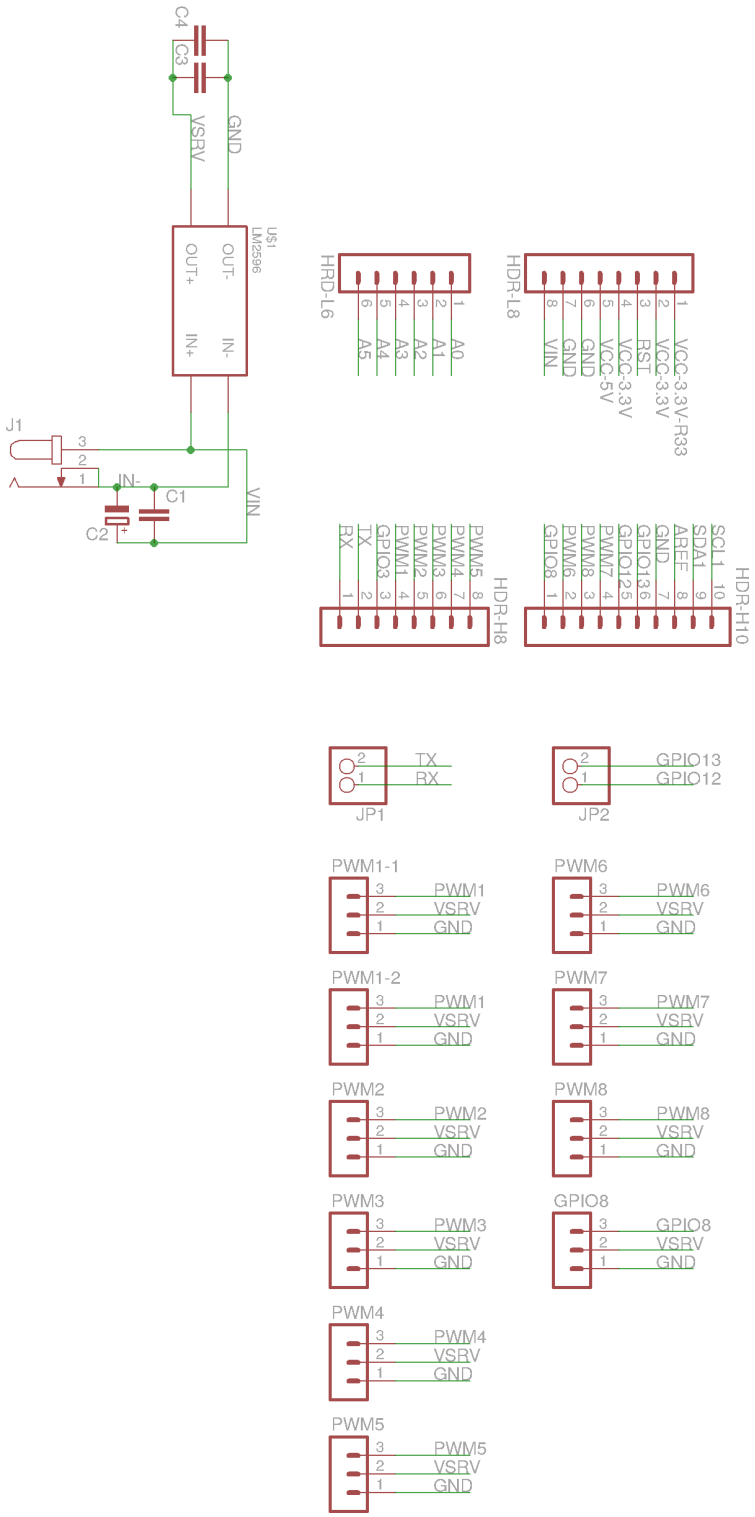
   ```
   UDOO$ cd RoboticArm
   ```

   a s využitím nástroje `qmake` si nechat vygenerovat Makefile, pomocí něhož celý projekt zkompilujeme:

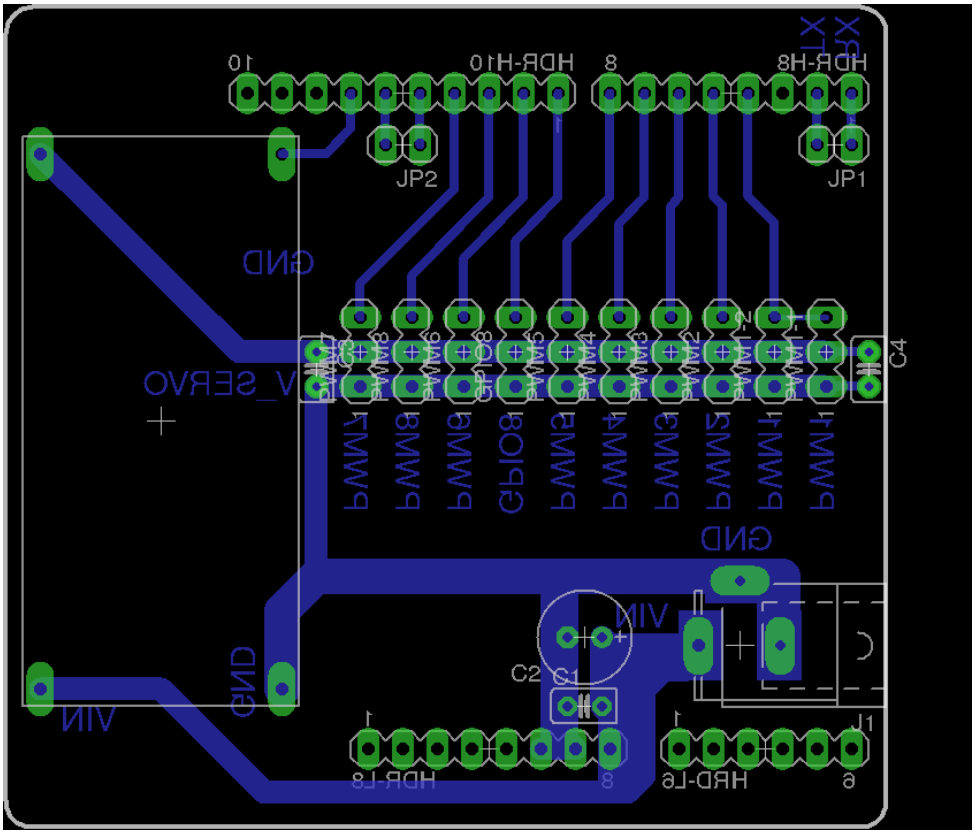
   ```
   UDOO$ qmake
   UDOO$ make
   ```

4. Samotná kompilace trvá na jádru Cortex A9 cca 5 minut. Po jejím skončení lze odinstalovat vývojové balíčky *libgstreamer0.10-dev* a *libopencv-dev*, které pro samotný běh aplikace nejsou třeba.

C Rozšiřující deska pro připojení serv



Obrázek 22: Schéma rozšiřující desky pro připojení serv.



Obrázek 23: DPS rozšiřující desky.